



## DATA DRIVEN LOOPS

Gregory R. Ruth

Key Words: Automatic Programming  
Software design  
Very high level languages

*This empty page was substituted for a  
blank page in the original document.*

Contents

I	Introduction.....	1
I.1	The HIBOL Language: A Brief Introduction.....	1
I.I.1	Flows.....	1
I.I.2	Flow Expressions.....	2
I.I.3	Flow Equations.....	3
I.I.4	Example.....	3
I.I.5	Additional Information.....	4
I.2	Iteration Sets and Explicit HIBOL.....	4
I.3	Implementation from a HIBOL Description.....	6
I.4	Data Driven Loops.....	7
II	Structure of Data Driven Loops.....	11
II.1	Loop Terminology.....	11
II.2	Kinds of Computations and Their Loops.....	11
II.2.1	Simple Computations.....	12
II.2.2	Matching Computations.....	14
II.2.2.1	Expressions Involving Flows with a Uniform Index.....	14
II.2.2.2	General Discussion of Expressions Involving Flows with Mixed Indices.	17
II.2.2.3	Mixed-Index Flow Expressions Allowed in HIBOL.....	18
II.2.3	Simple Reduction Computations.....	24
II.2.4	Aggregate Computations.....	28
II.3	General Loop Structure and Description.....	28

II.3.1 Formal Representation of Nested Loop Structures.....	29
II.3.2 Computation Implementation.....	30
II.3.2.1 Level Position of I/O and Calculations.....	30
II.3.2.2 Position of I/O and Calculations Within Their Assigned Levels.....	31
II.3.3 Examples.....	32
III Computation Aggregation and Loop Merging.....	36
III.1 Loop Aggregability.....	36
III.1.1 Level Compatibility Between Loops.....	37
III.1.2 Order Constraint Compatibility Between Loops.....	38
III.2 Merging Loops.....	41
III.3 Non-Totally-Nested Loops.....	42
III.3.1 Example 1: Aggregating Computations with Incompatible Order Constraints..	43
III.3.2 Example 2: Aggregating Computations That Are Not Level-Compatible.....	44
IV Driving Flow Sets.....	46
IV.1 A Theory of Index Sets and Critical Index Sets for Data Driven Loops.....	47
IV.1.1 Definitions and Useful Lemmas.....	47
IV.1.2 Critical Index Set Theorems for Computations.....	48
IV.1.3 Examples.....	52
IV.1.4 Driving Flow Set Sufficiency.....	55
IV.1.5 Minimal Driving Flow Sets.....	57
IV.2 Determination of Index Set Inclusion.....	58
IV.2.1 Characteristic Functions for Index Sets.....	59
IV.2.1.1 Variables.....	61

IV.2.1.2 (DEFINED variable-reference).....	62
IV.2.1.3 Correspondence Between Logical and Set Theoretic Notations.....	62
IV.2.2 Back-Substitution of Characteristic Functions.....	64
IV.2.3 Example.....	65
V Loop Implementation in a High Level Language (PL/I).....	68
V.I Single-Level Loops.....	68
V.I.1 Simple Computations.....	68
V.I.1.1 Necessary Data Objects and Their Declaration.....	69
V.I.1.2 Loop Initialization.....	70
V.I.1.3 EOF Checking and Loop Termination.....	70
V.I.1.4 The Loop Itself.....	71
V.I.2 Uniform-Index Matching Computations.....	71
V.I.3 More Than One Driver--Active Drivers.....	74
V.2 Multiple-Level Loops.....	77
V.2.1 Reduction Computations.....	78
V.2.2 Mixed-Index Matching Computations.....	80
V.3 Aggregated Computations.....	81
V.4 Aggregated Flows.....	84
V.5 Access Methods and Their Implementations.....	86
V.5.1 Sequential Access.....	88
V.5.2 Core Table Access.....	88
V.5.3 Random Access.....	89
V.6 The General Case--A Summary.....	91

Appendix I: The Simple Expositional Artificial Language (SEAL).....	92
References .....	95
Index .....	96

## Part I: *Introduction*

### I.I The HIBOL Language: A Brief Introduction

The notion of the data driven loop arises in connection with our work in the Very High Level Language HIBOL and the automatic programming system (ProtoSystem I) that supports it. Although the concept is of general interest outside of VHLL's and automatic programming, we find it profitable to use HIBOL as a vehicle for our discussion and a means of narrowing the scope of our discussion. Therefore we first present a brief description of the domain which HIBOL treats.

#### I.I.I Flows

The HIBOL language concerns a restricted but significant subset of all data processing applications: batch oriented systems involving the repetitive processing of indexed records from data files. It provides a concise and powerful way of dealing with data aggregates. HIBOL has a single data type, the *flow*. This construct is a (possibly named) data aggregate and represents a collection of uniform *records* that are individually and uniquely indexed by a multi-component *index*. The components of a flow's index are called *keys* and the set of an index's keys is called its *key-tuple*.<sup>1</sup> Each record has a single data field (*datum*) in addition to the index information. (Real-world data aggregates, such as files, with more than one datum per logical record are abstracted in HIBOL as separate flows, one for each data field.)

---

<sup>1</sup> This term is historical. A more expressive term would be "key set", but that has historically been used to indicate the universe from which a key may take its values.

### 1.1.2 Flow Expressions

*Flow expressions* can be formed through the application of arithmetic operators such as "+" or "\*" to flows. The meaning of such an application to two flows is that the operation is applied to the data of corresponding records (those with matching indices) of the argument flows. The result is a new flow, having a record for each matched pair for which the operation was performed. The index value of such a record is identical to that of the matched pair, and the datum value is the result of the operation performed on the data of the pair. This concept is generalized to an arbitrary number of flow arguments.

Flow expressions can also be constructed using a conditional operator (similar to a "CASE" statement) which evaluates logical expressions in terms of corresponding flow records in order to select and then compute an expression as the individual records of the flows are processed. The logical expressions are constructed using the arithmetic comparison operators ">", "<", and "<". In addition the PRESENT operator may be used to test the presence of a record in a flow for a given value of the index of that flow. These may be composed using the logical connectives "AND", "OR" and "NOT".

Finally, there is a class of reduction operators permitted on flows and flow expressions. The function of such an operator is to reduce a flow with an n-key index to one with an m-key index, where  $m < n$ , and the key-tuple of the m-key index is a subset of the key-tuple of the n-key index. All records of the argument flow that correspond to a single record of the result form a set to which a reduction operator (e.g. "maximum", "sum") can be applied to obtain a single value.

### I.I.3 Flow Equations

Relationships between flows are expressed by *flow equations* of the form:

<flow-name> IS <flow-expression>

where <flow-name> is a named flow and <flow-expression> is a flow expression in terms of named flows. The right- and left-hand sides must have identical indices.

### I.I.4 Example

Consider a chain of stores whose items are supplied from a central warehouse. The collection of store orders for item restocking on a given day can be thought of as a flow called, say, CURRENTORDER. A record of that flow contains the quantity ordered by a particular store of a particular item. Each record has as its datum the quantity ordered and a 2-component index identifying the store making the order and the item ordered (the keys of the index are a store-id and an item-id). Let BACKORDER be the name of a flow (of similar structure) representing the collection of (quantities of) previous orders that could either not be filled or filled only partially.

The HIBOL statement

DEMAND IS CURRENTORDER + BACKORDER

describes a new flow DEMAND representing the total demand of each item by each store. That is, each record in DEMAND contains a 2-component (item-id, store-id) index identifying its datum which is the sum of the data for the same item and store in the CURRENTORDER and BACKORDER flows.

The HIBOL statement

ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

illustrates the use of the reduction operator SUM. It describes a new flow ITEMDEMAND representing

the total demand of each item from *all* stores. That is, each of its records has a single-component index (item-id) identifying a particular item; and its datum is the total quantity in demand summed across all stores in the chain.

### 1.1.5 Additional Information

The computational part of a data processing system can be described by giving a full set of flow equations of the type shown above. To complete the system's description additional data and timing information must be given:

- for each flow, the components of its index, the type of its data value, and the periodicity with which it is computed
- for each key its type
- for each period its time relation to other periods

### 1.2 Iteration Sets and Explicit HIBOL

A flow expression, as explained above, represents a set of records obtained by the record-by-record application of a formula to the records of the flows that appear as terms in the expression. In this paper we shall be interested in exactly for which index values (and thus records) the indicated formula is applied. The set of these index values is termed the *iteration set*<sup>2</sup>

The HIBOL language is rather informal about specifying iteration sets. It contains abundant provisions (through the use of defaults) for implicit semantics based on the presence or absence of records in the flows appearing in flow expressions. For example, the HIBOL flow expression

CURRENTORDER + BACKORDER

---

<sup>2</sup> After Baron [1].

describes a flow that has a record for each index value for which either CURRENTORDER or BACKORDER (or both) has a record:

if both flows have a record for a given index value, the resultant flow has a record with the same index value, whose datum is the sum of those of the corresponding records in the two flows;

if only one flow has a record for a given index value, the resultant flow has a record with the same index value and the same datum value;

otherwise there is no record in the resultant flow.

One way of looking at the semantics of addition in HIBOL, then, is to convene that the operation

**+ is performed if and only if at least one of its operands is present and that each missing operand is treated as if it were the additive identity (0).**

Although such conventions are convenient in writing HIBOL, for the sakes of clarity and rigor, we require fully explicit iteration set specifications. Such can be obtained through the thorough use of the HIBOL primitives IF and PRESENT. Thus, the fully explicit form of the above HIBOL flow expression would be:

CURRENTORDER + BACKORDER	IF	CURRENTORDER PRESENT AND BACKORDER PRESENT
ELSE CURRENTORDER	IF	CURRENTORDER PRESENT
ELSE BACKORDER	IF	BACKORDER PRESENT

Here the index values for which the flow expression's formula is to be applied have been made explicit by restructuring it as a three-clause conditional expression in terms of three sub-expressions, each of whose iteration sets is specified by an associated condition on the presence of records in the flows involved. This is a legal HIBOL flow expression, although in view of the existing conventions it is overspecified (redundant). For our purposes we will distinguish a

language called Fully Explicit HIBOL (FE-HIBOL) where legal utterances are one subset of the legal utterances of HIBOL in which the iteration set of each flow expression is **empty**, and **expressions**

~~are qualified by the condition that all of the flow expressions in them that contain~~ **flow expressions** ~~are~~ **empty**.  
In general what this means is that no flow expression in FE-HIBOL is legal unless it is (at least) qualified by the condition that all of the flow expressions in them that containing are

~~empty~~ **empty** **flow expressions** in FE-HIBOL.

~~empty~~ **empty** **flow expressions** in FE-HIBOL.

(1) A

~~empty~~ **empty** **flow expressions** in FE-HIBOL are predicted to have an O

(2) A/B **IF B PRESENT**

~~empty~~ **empty** **flow expressions** in FE-HIBOL are predicted to have an O

(3) A/B **IF A > 40**

~~empty~~ **empty** **flow expressions** in FE-HIBOL

Their correct versions would be:

~~empty~~ **empty** **flow expressions** in FE-HIBOL are predicted to have an O

(1) A **IF A PRESENT**

~~empty~~ **empty** **flow expressions** in FE-HIBOL are predicted to have an O

(2) A/B **IF A PRESENT AND B PRESENT**

~~empty~~ **empty** **flow expressions** in FE-HIBOL are predicted to have an O

(3) A/B **IF A PRESENT AND B PRESENT AND A > 40**

Throughout the rest of this paper, unless explicitly stated otherwise, all HIBOL expressions will be

written in FE-HIBOL

### 1.3 Implementation from a HIBOL Description

The implementation of a HIBOL description of a data processing system involves three basic stages:

1. Data Analysis: to extract the technique language needed to do the implementation of the system described.

2. Code Generation: to generate the code needed to do the implementation of the system described.

The HIBOL description must be understood in data processing terms. The HIBOL

must be work in accordance with HIBOL layer 2 of the Prolog language.

In Prolog System I, in fact, immediately after a HIBOL data processing system is described it is translated into an internal language (DSSL) which has exactly this requirement.

description is *declarative* in nature: it describes the relationships among the flows. An implemented data processing system is *procedural* in nature: it must describe in detail how the flows are computed. The flow equations must be reinterpreted as basic computation steps (with an output flow and one or more flows as inputs) and constraints on the order in which these computations can be performed (the computation producing a flow must be performed before any computations using that flow) must be made explicit.

#### Design:<sup>4</sup>

The implementation will make use of files of data to be processed by job steps which will in turn create other files. Each file will contain the information represented by one or more flows; each job step will perform the processing to satisfy one or more flow equations. The design of each file (information contained, organization, storage device, record sort order) and of each job step (equations implemented, loop structure, accessing methods used) should be made in such a way as to minimize some overall cost measure (e.g. dollars-and-cents cost, time used, number of secondary storage I/O events) for the execution of the data processing system. Typically this requires dynamic (behavioral) analysis of tentative design configurations.

#### Code Generation:

The system's design must be coded in a supported high-level language so that it can be executed.

### 1.4 Data Driven Loops

Each flow equation represents a *computation* whose implementation is essentially iterative in

---

<sup>4</sup> In ProtoSystem I the design process is performed by the Optimizing Designer module.

systems. That is, the relationships between data and values of variables are mapped by the flow(s) in the flow expression on the right-hand side, performing the indicated operations in order to generate the records of the flow on the left-hand side. In many programming languages this iteration is effected through the use of a loop construct, like (loop as x from 10 to 100 by 10). A single loop (possibly with nested loops) can produce a flow expression. That is, a loop can be devised that will produce the entire flow appearing on the left-hand side of the flow equation from the flows appearing on the right-hand side. In functional terms, the flows on the right-hand side of the flow equation are called the **inputs** to the corresponding loop and the flow appearing on the left-hand side is the **output**.  
 Thus, The body of the loop specifies the computation of a given value. A record of the output is to be generated and, in systems, for example, **ANSWERING EXPRESSIONS OF TIBOL**,  
**expressions** data in atom ad intord (bsau abdutn gntzgods, stutusla qpol, beinmeiqml znoturpe)  
 (subroutines to **16** atoms, bsa ad 16), **NO A PRESENT** will be performed. If no value is obtained of  
**ELSE A IF A PRESENT**  
 then **NO A PRESENT** will be performed. (atoms: gntzgods stab ad to (atmvs C,f, gntzgods))  
 The body of the corresponding loop will distinguish two cases and corresponding courses of action:  
 knowing that **NO A PRESENT** is satisfied to **Answer** (let's say **ans**)

1. Records in both A and B are present for the current value of the loop index, in which case a corresponding record of the flow S is produced where **ans** is the sum of the values in the records of A and B.
2. Only A contains a record corresponding to the current value of the index, in which case a corresponding record of S is produced that is identical to A's record.

If neither of these cases obtains, no output record is produced.

13.00.1 mod 0. chd 1.1

Clearly, in a correct implementation the body of the loop must be performed for every index of **NO A PRESENT** if no value is obtained. We call this value for which records of the input(s) exist that will be used to produce an output record. We call

---

<sup>5</sup> APL, due to its support for various vectorial operations, is often used as a language for **vectorized** **VLISP**.

any set of values for a particular index an *index set* and we distinguish two special kinds of index sets:<sup>6</sup>

The set of index values for which a flow  $F$  contains a record is called the *index set of  $F$*  (denoted  $IS(F)$ ).

The set of index values for which an input flow  $F_i$  contains a record that will be used in generating a record of the output flow  $F$ , the *critical index set of  $F_i$  with respect to  $F$*  (denoted  $CIS_F(F_i)$ ).

These two should not be confused.  $CIS_F(F_i)$  for some flow  $F$  will often be a proper subset of  $IS(F_i)$ .<sup>7</sup>

The problem we face is that of finding some way of enumerating the critical index sets of each input so that loop can be properly driven.<sup>8</sup> It is generally impractical to use the set of all possible (legal) index values for which an input might have a record. For one thing this set may be unbounded. Even if it is finite and enumerable, it will often be much larger than the critical index set and thus grossly inefficient. In the DEMAND flow equation example given above, for instance, the critical index set of the input flow CURRENTORDER is likely to be orders of magnitude smaller than its maximum possible size (the case where every store has orders for every item).

A much more efficient way of enumerating a set of index values that is assured to cover the critical index sets of the inputs is to use the union of the index sets of the input flows. This will work because a record of the output can be produced only if there is some input flow in which that

---

<sup>6</sup> Unfortunately, this terminology is at variance with that used by Baron in his thesis [1]. Baron uses the term "critical index set" to mean what we call the "index set".

<sup>7</sup> On no account, of course, can it be other than a proper or improper subset of  $IS(F_i)$ .

<sup>8</sup> This statement is somewhat oversimplified, but it will suffice for now. A fully precise statement of the problem is given by the Fundamental Driving Constraint in Part IV.

record is present. Moreover, the nature of these sets is easily understood simply by reading the input flows (which have to be read anyway). A loop that is thus driven by index values supplied by its inputs is said to be a *data-driven loop* and the indices used under one loop are termed its *driving flows* (the set of flows that drive a loop are called its *driving flow set*).<sup>(7.21)</sup> The structure and implementation of data-driven loops is the subject of this paper.<sup>(7.22)</sup> It is also noted that the driving flow set of a loop is the union of all the driving flow sets of the loops it contains.

It will not always be necessary to read all inputs to implement data-driven index sets. From the flow equation for  $S$  above it can be deduced that the index set of  $A$  alone is sufficient to cover  $CIS_0(A)$  and  $CIS_1(A)$ . This is so because indexable data items in  $S$  will be needed only if there is corresponding record in  $A$ . Therefore, the loop can be driven by  $A$  alone; that is, it is sufficient to perform the body of the loop only for those values of  $i$  where  $A[i]$  has a record. In general, for the sake of efficiency it is of interest to find the minimum set of subsets of the inputs that will be sufficient to drive the loop. Again, such an index set need not be unique.

Example 7.23: Consider the flow diagram in Fig. 7.10. It determines the sum of the elements of a vector  $A$  given in the form  $A[1:n]$ . The algorithm consists of two nested loops. The outer loop iterates over  $i$  from 1 to  $n$ , and the inner loop iterates over  $j$  from 1 to  $m$ . The inner loop adds the value of  $A[i,j]$  to the sum. The outer loop adds the value of  $S$  to the sum. The flow diagram is as follows:

```

    S := 0
    for i = 1 to n do
        S := S + A[i]
        for j = 1 to m do
            S := S + A[i,j]
    endfor
endfor
    
```

(7.21) A record set is called a *driven set* if it has been read at least once. The term "driven" is derived from the fact that the record set is "driven" by the record set of the record set. The record set of a driven set is called the "driving set".<sup>(7.22)</sup> On the other hand, a record set is called a *driving set* if it has been read at least once. The record set of a driving set is called the "driven set".<sup>(7.23)</sup>

## Part II: Structure of Data Driven Loops

Before a general treatment of data driven loops can be developed it is necessary to examine the structures of the loops encountered in the HIBOL system. We begin by presenting a taxonomy of computation types and their corresponding loop implementations.

### II.1 Loop Terminology

Before discussing loop structures it is useful to establish some terminology. By the term *loop* we mean a control construct which somehow enumerates a set of values for a *loop-index* and which performs a fixed sequence of statements (its *body*), once for each value of the loop-index. A loop may contain one or more loops within its body. The inner loops are said to be *nested* within the outer (enclosing) loop and the structure as a whole is called a *nested loop structure*. Each enclosure defines a different *level* of the nested loop structure. The degenerate case of a nested loop structure, where there is no loop in the body of the outer loop, is called a *single-level loop*, since there is only one loop level.

A *totally nested loop* is a nested loop structure whose component loops are totally ordered under enclosure (i.e. for any two loops  $L_1$  and  $L_2$  either  $L_1$  is inside  $L_2$  or  $L_2$  is inside  $L_1$ ).

### II.2 Kinds of Computations and Their Loops

Each run (computation, job step, program) in the implementation produced for a HIBOL description of a data processing system is essentially a loop that iterates over the records of its input files to generate records of its output file(s). The structure of this loop depends on the nature of the computation being performed. We will begin with computations that directly implement single HIBOL flow equations of various types. Then we will consider computations that implement more than one flow equation (aggregated computations) simultaneously.

### II.2.1 Simple Computations

Input record map to statement II.164

A simple computation computes one expression that depends on having a single flow. Thus, it takes one flow as input and produces one flow as output. A simple computation's computation is described by the HINQOL flow equation  $\text{PAY IS } \text{HOURS} * 3.00$

PAY IS	$\text{HOURS} * 3.00$	IF HOURS PRESENT AND
		$\text{HOURS} > 40$

colon(1,1) and 1.1

~~IF PAY IS 120 + (HOURS - 40) \* 1.5 ELSE HOURS \* 3.00~~

This assumes that there is a flow HOURS indicating employment units or hours worked, such as 45, datum the number of hours the individual employee worked. Each employee is paid \$3.00/hour, with time-and-a-half for overtime. PAY is the corresponding flow from pay to each employee who worked (i.e. had a corresponding record in it), assuming the goal (goalform) is function.

The flow implements this kind of computation in the simplest fashion by creating a local loop having a single input file, which is the driver. On each iteration the input of the loop is read and used to provide both a datum and index value for the body, which collects the one datum of the corresponding ~~value~~ record, combining the record form, condition and the current index value, and outputs it.

In the SEAL language (see Appendix I) this would look like:

~~goal THAT has been converted to chart 2.11~~

JG.111 a col branch to next/terminal set of (paying rate for nonstuffed) but next  
input is to choose or/also select just one of all records at index position, which is to collect  
set. To extract set no choice of you will be substance of T. (if it is true all to choose because of off  
sign) from sign (parent) test should happen after regard flow new .banished. tested and requires  
parent to be off and option chosen free sw and T. (goal converter to profit) is now (D2/H  
statement (notstuffed) before sign) notstuffed with the next

```

for each (employee-id) from HOURS

    get HOURS(employee-id)

    PAY(employee-id) =

        if defined(HOURS(employee-id))
            and not (HOURS(employee-id) > 40)

                then HOURS(employee-id) * 3.0

            else if defined(HOURS(employee-id))

                then 120.0 + (HOURS(employee-id) - 40) * 4.5

            else undefined

        if defined(PAY(employee-id))
            then write PAY(employee-id)

    end

```

The for-end construct represents the basic iteration over values of the index employee-id. It specifies that the values for the index are obtained from the HOURS flow. For each index value, the corresponding record of HOURS is read, the corresponding record of PAY is generated, and (if generation was successful) that record is written out. Notice that the PAY calculation is a direct translation from the HIBOL flow equation.

For reasons of exposition the loop implementation presented here is of the most general form.

An actual implementation would incorporate various efficiency enhancing improvements.<sup>9</sup> Nevertheless, we shall continue to use such forms to show explicitly where I/O and testing occur conceptually.

---

<sup>9</sup> For instance, since the for has to read the next record of the driver to get the current index value, the get could be omitted. Furthermore, the defined tests in the PAY calculation could be omitted since they are testing the presence of record which must be present. Finally, in this computation, the check before output could also be omitted.

### II.2.2 Matching Computations

A matching computation computes a non-reduction flow expression involving two or more flows. Thus it is similar to a simple computation, but instead of operating on a single record of a single input flow to produce an output record, it operates on a set of corresponding records, one from each input flow. Correspondence is established by common index values. The name "matching computations" derives from the necessity of matching up the records of the inputs by index values before they can be operated on.

Two sub-classes of matching computations can be distinguished depending on whether all of the inputs have indices with identical key-tuples or not.

#### II.2.2.1 Expressions Involving Flows with a Uniform Index

Consider the a pay calculation similar to that given above, but where employees are paid various hourly rates. Let RATE be a flow, indexed by (employee-id), each of whose records has as its datum the hourly pay rate for the employee indicated by its index value. The pay calculation then becomes

```
PAY IS HOURS * RATE
IF      HOURS PRESENT
AND RATE PRESENT
AND NOT HOURS > 48
ELSE
  RATE * 48 +
  (HOURS - 48) * 1.5 * RATE IF      HOURS PRESENT
                                AND RATE PRESENT
```

HOURS and RATE have identical indices, each consisting of the single key "employee-id". The loop that implements such a computation has a single level.

Because a record of the output is generated only if there is a record in the HOURS file, that

file alone is sufficient to drive the loop. (Alternatively, by similar reasoning, the RATE file could be used to drive the loop.) This is the simplest case of a matching computation because only one input is needed to drive the loop. (The computation of the flow S above is also of this type.) On each iteration the next record of the HOURS file is read, the corresponding RATE record is fetched, and the computation of gross pay performed.

This loop is represented in the SEAL language thus:

```

for each {employee-id} from HOURS

    get HOURS{employee-id}

    get RATE{employee-id}

    PAY{employee-id} =

        if defined[HOURS(employee-id)]
            and defined[RATE(employee-id)]
            and not(HOURS(employee-id) > 40)

                then HOURS(employee-id) * RATE(employee-id)

            else if defined[HOURS(employee-id)]
                and defined[RATE(employee-id)]

                    then RATE(employee-id) * 40 +
                        (HOURS(employee) - 40) * RATE(employee-id) * 1.5

                else undefined

        if defined[PAY(employee-id)]
            then write PAY(employee-id)

end

```

Again, the defined checks on the driver, HOURS, are superfluous. But those on RATE are necessary (to determine whether the corresponding get was successful) and the defined check on PAY is necessary (so that a record is written if and only if a datum was generated).

Now consider the HIBOL flow equation for the DEMAND flow given above:

ed value. **DEMANDS** will again be implemented as **ITEMS**. **IF CURRENTBTER PRESENT AND BACKBTER PRESENT** (goal will switch to **ITEMS**) **IF CURRENTBTER PRESENT** (goal will switch to **ITEMS**) **ELSE CURRENTBTER** **IF CURRENTBTER PRESENT** (goal will switch to **ITEMS**) **IF BACKBTER PRESENT** (goal will switch to **ITEMS**) **ELSE BACKBTER** **IF BACKBTER PRESENT** (goal will switch to **ITEMS**). Again, **CURRENTBTER** and **BACKBTER** have identical indices, both consisting of the compound keys item-id and state-id, so again the implementing loop has a single level. In this case both **ITEMS** and **ITEMS** implement **ITEMS**.

When there is more than one driver for a computer some of the drivers will have records  
for a given index while the others do not.<sup>10</sup> If the drivers have their records sorted in the same  
order (say, alphabetically) by index values, the loop may be performed by making only one pass  
through the inputs in the following manner:<sup>11</sup>

**Q. Read the first sound of each word.** (18) < (bi-ee-yo-toms) (RUDH) Jon. bins

**LIBI-5001(g)(1) TARIHEK 12b bns**

Re: 1 (b) - equal loss of 25 MW) log bus

1. Use the smallest index within `minIndex` that `MinIndex <= logn` and `logn <= minIndex + perLogBody`. Fetching other non-driver records if necessary.  

$$(\text{fb1-asyc0(qw0)} \text{STAR} \text{fb1-be1}) \text{lab-11-sale}$$
  2. Discard all driver records until the entire `fb1-asyc0(qw0)` has been read, then read the next `perLogBody` records for every driver whose record was discarded.  

$$+ 84 + (\text{fb1-asyc0(qw0)} \text{STAR next})$$
  3. Report from `fb1-asyc0(qw0)` `STAR`  $\neq$  `(84 - (\text{fb1-asyc0(qw0)} \text{STAR}))`

If the only sorting constraint on the inputs is that the values be sorted in the same order (i.e. no constraints on the order of the records in the other fields) we can use a global hash table (one page for each non-driving input) to map from the driving input to the record.

or by search; but if any non-driving input is sorted in the same order as the drivers, its records can  
be fetched by sequential reading, which is generally more efficient.

If this were not the case, the two parts of the hypothesis would not be consistent with each other.

"If they are not similarly sorted, some inefficient moves must be used.

produces nearly zero DMM20 soft tail noise-ups with TORCH still retaining wolf

These details are implicit in the SEAL representation of the loop which is simply:

```
for each (item-id, store-id) from CURRENTORDER, BACKORDER  
    get CURRENTORDER(item-id, store-id)  
    get BACKORDER(item-id, store-id)  
    DEMAND(item-id, store-id) = ...  
    if defined(DEMAND(item-id, store-id))  
        then write DEMAND(item-id, store-id)  
end
```

### II.2.2.2 General Discussion of Expressions Involving Flows with Mixed Indices

The treatment of mixed-index flow expressions in this paper will be restricted to those that are legal in HIBOL. The restrictions that HIBOL imposes are made for good reasons. A brief discussion of the various conceivable types of mixed-index flow expressions is presented here in order to show the motivation behind these restrictions.

The various cases where the flows in a flow expression have mixed indices (i.e. their indices have different key-tuples) can be distinguished by the set interrelationships among the key-tuples.

Consider the case where flows have disjoint key-tuples (e.g. (w, x) and (y, z)). Correspondence among records of such flows is meaningless, so we do not allow them to appear in the same flow expression.

Now consider the more general case where there is intersection among index key-tuples, but the union of their pair-wise intersections is not identical to their (simple) union. In this case correspondence is always ambiguous. For example, consider the two flows: A with index (x, y) and B with index (y, z). Suppose that there are records in A for the particular index values (x<sub>1</sub>, y<sub>1</sub>) and

$(x_2, y_1)$  and that there are records on B for index values  $(y_1, z_1)$ ,  $(y_1, z_2)$  and  $(y_1, z_3)$ . Which of A's records correspond to which of B's records?<sup>12</sup>

For correspondence to be meaningful and unambiguous it must be the case that the union of the pair-wise intersections of the key-tuples of the indices involved is identical to their union. This is always the case when there exists an index among the flows involved whose key-tuple is a superset of all the key-tuples of the other flows.

To be sure, there are other ways of satisfying the condition of the preceding paragraph. These involve conjunctions of three or more indices. Consider, for instance, the three flows: A with index  $(x, y)$ ; B with index  $(y, z)$ ; and C with index  $(x, z)$ . Corresponding triplets are all unique and unambiguous, of the form  $(x_i, y_j, z_k)$ ,  $(y_j, z_k, x_i)$ ,  $(x_i, z_k, y_j)$ . For the sake of simplicity, however, this case is prohibited in HIBOL.

### II.2.2.3 Mixed-Index Flow Expressions Allowed in HIBOL

It is possible in HIBOL to apply operators to two or more flows having different indices as long as each index is a sub-index of the index of some unique flow involved (i.e. as long as the key-tuple of each index is a subset of the key-tuple of the index of the unique flow). Clearly, the index of this unique flow is identical to the index of the flow expression as a whole. HIBOL allows a mixed-index flow expression only if its computation can be driven by the set of those flows involved having indices identical to that of the flow expression.

---

<sup>12</sup> Of course, we could allow all pairs to match (in Cartesian product fashion) so that the expression A + B would represent the six possible combinations of additions for these 5 index values; but this would change (extend) the semantics of HIBOL.

For example, suppose we want to calculate the extended prices<sup>13</sup> of the current store orders (the flow CURRENTORDER) in our store chain example. Let PRICE be a flow indexed by (item-id), each of whose records has as its datum the per-item price associated with the item identified by its index. The flow equation for EXTENDEDPRICE, indexed by (item-id, store-id) would be expressed in HIBOL thus:

```
EXTENDEDPRICE IS CURRENTORDER * PRICE IF CURRENTORDER PRESENT
AND PRICE PRESENT
```

The intent here is: for every record in CURRENTORDER find the corresponding record in PRICE and, if the latter is present, multiply their respective data to calculate the datum of a corresponding record in EXTENDEDPRICE. Notice that because PRICE and CURRENTORDER have different indices ((item-id) and (item-id, store-id), respectively) the notion of correspondence must be extended in a natural way from pure identity of index values. We convene that for a particular value of item-id the index (item-id) matches any index (item-id, store-id) with the same value of item-id, regardless of the value of store-id. This augmented definition of correspondence is extended to the general case where the key-tuple of one index is a subset of the key-tuple of another. That is, for given values of  $k_1, \dots, k_m$  the index  $(k_1, \dots, k_m)$  is said to match any instance of an index  $(k_1, \dots, k_m, k_{m+1}, \dots, k_n)$  with the same values of  $k_1, \dots, k_m$ , regardless of the values of  $k_{m+1}, \dots, k_n$ .

Since a set of input flows, each with index identical to the flow expression's, can be used to drive a mixed-index matching computation, its implementation is similar to that for a uniform-index matching computation: the sorted drivers are read in such a way as to enumerate the critical index sets of all of the input flows; the resulting index values are used to fetch records from the rest of the inputs (including all those whose indices are sub-indices of the flow expression's index).

---

<sup>13</sup> The extended price of a quantity ordered is the product of the quantity and the per-item price.

The general form SEAL implementation is single-threaded as shown below (signature omitted)

```

    /bin/for <each, index> from <cursor> -> <cursor>
        <cursor>.get <input>_1
        <cursor>.get <input>_2
        ...
        <cursor>.get <input>_n
    
```

HTTPD

**THE output of <cursor> ...** is  $\{ \langle \text{item-id} \rangle : \langle \text{store-id} \rangle \}$

$\langle \text{item-id} \rangle$        $\langle \text{store-id} \rangle$

**if defined output(<index>)**

bus 30189 in **then** with output(<index>) <cursor>.get <input>\_1 to <cursor>.get <input>\_n

**else** <cursor> is to ensure sets of stores or sets of subsequent items by defining <cursor>.set. If <cursor>.set is not defined, it will cause compilation error. **else** **use** **it** **to fetch** **the** **records** **of** **the** **new** **driven** **activity** **under** **the** **current** **activity** **as** **either** **(** **bi-m1** **)** **bus** **(** **bi-m2** **)**.

**iii) Add More I/O examples** can be written by adding more **cursor** to **the** **cursor** **or** **the** **flow** **expression** **driven** **by** **the** **additional** **cursor** **as** **next** **cursor** **in** **<cursor>** **records** **sorted** **by** **its** **sub**/**subsub** **activities**. It corresponds to multiple **fetching** **and** **reusing** **set** **to** **compute**.

**Consider:** the **EXTENDER** **computation** is facilitated by the **CURRENTORDER** **which** **has** **(** **item-id**, **store-id** **)** **as** **its** **index**. The **index** **of** **the** **line** **(** **item-id**, **PRICE**, **target-store-id** **)**. If the records of both **CURRENTORDER** are sorted by **item-id**, then all of the **CURRENTORDER** records for a given value of **target-id**, can be processed in sequence without reading back into **PRICE**. On the other hand, if the records of **CURRENTORDER** are not sorted by **item-id**, then by **process**, the processing of the each of its records will require a fresh **fetch** from **PRICE** which is a **public** **table** **of** **PRICE** **so** **that** **table** **to** **be** **read** **more** **than** **once**. because this **table** **will** **contain** **only** **two** **lines** **and** **it** **is** **not** **possible**.

In the case where the records of **CURRENTORDER** are sorted by **item-id**, the **EXTENDER** **computation** **is** **implemented** **as** **a** **nested** **loop** **structure** **with** **inner** **loop** **(** **one** **inside** **the** **other** **)**.

Basically, the outer loop chooses a value of the sub-index (item-id) and fetches the corresponding PRICE record. Then it performs the inner loop. Within the inner loop the value of the item-id key is held constant. All corresponding records of CURRENTORDER are read and the computation described in the flow equation is performed using the data of these records together with the datum of the PRICE record fetched in the outer loop. The results are used to build and output the corresponding records of EXTENDEDPRICE. This process is repeated until the flows are exhausted.

In detail the implementation is as follows. Before either loop is entered a record of CURRENTORDER is read. The outer loop uses this record to obtain the first value of the sub-index (item-id) and fetches the corresponding record from PRICE. Then it performs the inner loop. The inner loop uses the current record of CURRENTORDER and continues to read records sequentially from CURRENTORDER until the sub-index is observed to change or an end-of-file condition occurs. When either of these conditions occurs, it exits to the outer loop. If an eof has occurred, the outer loop exits. Otherwise it iterates, using the sub-index value of the current CURRENTORDER record as the new value to be held constant in the inner loop, fetching the corresponding PRICE record and performing the inner loop again.

The corresponding SEAL code is:

```

for each (item-id) from CURRENTORDER

    get PRICE(item-id)

    for each (store-id) from CURRENTORDER(item-id)

        get CURRENTORDER(item-id, store-id)

        EXTENDEDPRICE(item-id, store-id) =

            if defined(CURRENTORDER(item-id, store-id))
                and defined(PRICE(item-id))

                    then CURRENTORDER(item-id, store-id) * PRICE(item-id)

            else undefined

            if defined(EXTENDEDPRICE(item-id, store-id))
                then write EXTENDEDPRICE(item-id, store-id)

```

Notice that the outer loop is driven by CURRENTORDER (the whole flow), but that the inner loop is driven by CURRENTORDER(item-id) (the sub-flow of CURRENTORDER consisting of just those records whose indices correspond to the value of the sub-index (item-id) fixed by the outer loop). What this means is that for the outer loop the next value of the sub-index (item-id) will be taken from the next record of the CURRENTORDER flow. But for the inner loop the next value for the sub-index (store-id) will be taken from the next record of the sub-flow of CURRENTORDER corresponding to the current value of (item-id); if there are no further records in CURRENTORDER for this fixed value of (item-id) this will be treated just like an end-of-file condition and the iteration of the inner loop will terminate. Thus the inner loop is driven by a succession of sub-flows, one for each iteration of the outer loop.

This nested-loop implementation scheme is easily extended to 3 or more loop levels when appropriate sorting constraints hold among the flows involved. For example, suppose that there

are 3 flows involved: A with index  $(k_1, k_2, k_3)$ ; B with index  $(k_1, k_2)$ ; and C with index  $(k_1)$ . And suppose further that B is sorted by  $k_1$  and that A is sorted first by  $k_1$  and, *within segments corresponding to a fixed value of  $k_1$* , the records of A are further sorted by  $k_2$ . Then the flow equation can be implemented using a nested loop structure involving 3 loops (innermost loop, middle loop and outermost loop). The outermost loop chooses a value for the key  $k_1$  to be held constant within the middle loop (and perform in the innermost loop, which is contained in the middle loop). It also fetches the corresponding record of C for use within the contained loops. Then it executes the middle loop, which, in turn, choose a value for the key  $k_2$  to be held constant within the inner loop. The middle loop also fetches the corresponding record of B for use within the innermost loop. Then it executes the innermost loop. In the innermost loop the values of the keys  $k_1$  and  $k_2$  are held constant. The innermost loop reads all corresponding records of A, using their data and those of the already read records to perform the calculations described in the flow equation and to build and output the records of the output flow. When the innermost loop has read and processed all records of A corresponding to the fixed values of  $k_1$  and  $k_2$ , it exits to the middle loop, which chooses a new value for  $k_2$  and iterates. When the middle loop has exhausted all possibilities for the value of  $k_1$  fixed in it, it returns to the outermost loop, which chooses a new value of  $k_1$  and iterates. This loop structure expressed in the SEAL language looks like:

For each  $(t, j)$  from  $\{1, \dots, T\} \times \{1, \dots, J\}$  xbmrlvw S  $\left( t, j, \varphi_{t, j}, \theta \right)$  xbmrlvw A  $\left( b m v l v n \right) \text{ and } S \left( t, j \right)$

**Exercice 6** : A ya tenté de voler à B deux billets de 10 francs. A a été arrêté et A et B ont été interrogés.

for graph  $G$  and form  $A$  that will go from  $A$  to  $\text{absorb}(A)$  and  $A$  to  $\text{subset}(A)$ , it is given that  $\text{absorb}(A) = \text{subset}(A)$ .

blended with 1% agarose gel from Bio-Rad (Agarose 3000, 1% agarose, 1x TAE). Gel electrophoresis was performed at 100 V for 1 h.

*...and all the best of the world.* **After** these bairns are set in school, poor laddies will have to go to work.

**«Our form collection»** b10231: este colecție este în calegrafie și tipărită pe hârtie.

The following table summarizes the results of the experiments.

the following section, we will present our experimental setup and discuss the results.

A la clausura se informó que las tres personas detenidas permanecen en la cárcel de Heredia.

and 200 mg/kg bw/day were included and consisted of chosen best wheat(s) and its second best since 2001.

### **H23 Simple Reduction Combinations**

Early growth is often slow, with the young taking up to about six months before they begin to move around.

A simple reduction computation computes a reduction flow expression involving a single set of  $\text{size } n \times 1$  bars,  $\lambda$  to satisfy boxset set of partitionables  $\Lambda$  to reduce the branching bar flow. A reduction flow expression is the application of a reduction operator to an arithmetic flow boxset(s) and goal ribbon set and  $W$   $\text{length } n \times 1$  bar,  $\lambda$  to satisfy wcn a records down goal ribbon expression. Often the flow expression to which the reduction operator is applied is simply a single wcn a records down goal transition set of entries if it is not boxset,  $\lambda$  to satisfy sets to partitionable its flow itself, as in the previously cited example.

**THE ENDLESS IS THE SUM OF BEING FOR EACH WORLD**

where  $S\Gamma$  is applied to  $\{E_{\alpha\beta}\}$ . It may be shown that

I TENDERED IS THE SON OF CONSTRUCTION + ENGINEERING  
FOR EACH WORK.

(written in HIBOL<sup>10</sup>) are also suitable. In our case the parameter of the reduction operator is

<sup>14</sup> The standard HIBOL form is used here for clarity and convenience; the corresponding FE-HIBOL form is rather cumbersome.

treated as a single flow.

Conceptually, the argument flow is partitioned into subsets (sub-flows) by an equivalence relation defined on the sub-index (a key or keys) indicated in the FOR EACH clause; then the reduction operator is applied to the members of each subset to generate the value of the datum of the output record corresponding to that subset. For instance, in the first example given above the DEMAND flow is conceptually partitioned into record subsets by item-id. Thus, all records in DEMAND whose index contains the value item-id<sub>1</sub> for the item-id key are in one subset, all records for item-id = item-id<sub>2</sub> are in another, and so forth (empty subsets are ignored). The datum for the record in ITEMDEMAND with index = (item-id<sub>i</sub>) is calculated by summing all of the data in the records in the subset corresponding to item-id = item-id<sub>i</sub>.

Conceptually, the implementing iteration for a simple reduction expression in a single flow consists of two loops, one nested inside the other. The inner loop implements the application of the indicated reduction operation to a subset of the input's records. Within this loop the value of the sub-index defining the subset is held constant. Returning to the SUM OF DEMAND example, the inner loop implements the summation of the data of the records of each subset of DEMAND. That is, the inner loop is performed for each value of item-id<sub>i</sub> for which there are records in DEMAND. Within the inner loop the particular value of the key item-id is held constant, all records of DEMAND corresponding to that key value are fetched and their data are summed.

The outer loop performs clerical work. It chooses a value the subsetting sub-index (e.g. a value of item-id), executes the inner loop (which fetches records of the input corresponding to the chosen sub-index and, for example, adds them to the accumulator), and when the inner loop is finished, it uses the resulting value as the datum of the output record corresponding to the chosen sub-index, and writes that record out.

for each item-id from 0 to 1000

**Bi-monthly budget of anthropogenic CO<sub>2</sub> release**

Consequently, the following will be considered as well: **sum = undefined**

for each *listens-idx* from **0** to **listens-idx**  
add to *listens* all nonempty root terms of *listens* obtained between *listens-idx* and *now*; to statement

www.english-test.net - free English tests and exercises

**and differentiates them - it starts - it, starts - it**

**QUESTION 1** ERKLÄREN SIE MEHRERE DÄCHER UND DÄCHER MIT DACHFENSTERN UND DACHGÄRten.

SWITZERLAND: The Swiss Federal Institute of Technology in Zurich has developed a new type of solar cell that is more efficient than conventional silicon-based cells.

QUANTITÉ de débit en litres par second et la valeur de l'écoulement dans lequel réagit cette quantité.

**becomes one single road bus before it goes away (as seen on your bridge test)**

The other two histories differ now. It seems as if all the members

end of an observation period until the chosen sample (below) goes from the category "in control" to "out of control".

af goed waren dit eerste bns (totaleind) en de volgende bns tot dat ze kunnen doorstaan.

recorde-se de que o presidente da República é o chefe do Poder Executivo, que exerce o governo e a administração.

and  
the former right-wing has taken up

It may at first seem unnecessarily baroque to initialize the accumulator sum to "undefined" in the outer loop, test it in the inner loop for definedness and then initialize it if undefined. In this simple example we could just initialize it to 0 in the outer loop and not bother with the definedness checks. We have chosen the former course for two reasons. First, we wish to make explicit the conditions under which the sum (and thus a record of the output ITEMDEMAND) is defined for a given value of the key item-id. Second, a little thought will show that for other reduction operations (viz. MAX and MIN) initialization of the accumulator must (at least conceptually) be postponed until the inner loop where the initializing value is obtained by the first get. Moreover, in general, when computations are aggregated (see below) and more than one activity is performed in the inner loop, it is then possible (if some driver besides DEMAND is used) that for some values of item-id no sum is calculated in the inner loop and thus sum is undefined on exit from that loop.

If the input flow is not sorted as above, the computation for a reduction operation becomes somewhat more complex. One possibility is to create and maintain separate accumulators for each value of the sub-index value occurring in the input flow. Since the number of accumulators cannot be known *a priori* (i.e. at compile time), storage for them must be allocated on the fly (during execution of the computation). In PL/I, for example, the following (roughly outlined) scheme might be used:

Declare an accumulator array to have CONTROLLED storage.

Make a pre-pass through the input flow to count the number of different sub-index values occurring.

Execute an ALLOCATE statement to define the size of the array.

Make a second pass over the input flow to perform the accumulation.

Write all accumulated values out to the output flow.

In this scheme there are two separate loops instead of a totally nested loop structure.

Alternatively, a nested loop, multi-pass scheme could be implemented. The outer loop would

determining the next sub-index value for which the current character is preceded by 16 zeros. It will then pass this value through the input file. The lower loop would then produce the concatenation for that sub-index value by reading through the file. This would result in 16 read times and 16 read writes.

**Various other reference test methods** (See [Section 2.1.1.1](#) for a listing of methods). **Interpretation** depends on what has been expected or measured previously (possibly by monitoring or testing previously).

We have chosen the former option for two reasons. First, we will be able to

**B24 Aggregate Components** (e.g., aggregate size to boost a audit base) must add double quotes around the value.

**Il Consiglio europeo ha approvato le norme che autorizzano l'Europa a prendere misure contro i paesi che violano i diritti umani**

If the upper five or six rows of teeth are missing, the lower teeth will be used to bite off the meat.

Georges D'Albigny was one of very few consumers he could identify.

### **1.3 General Land Surveying and Planning**

Now that we have officially discussed the kinds of data that may be available, we will focus on the specific kinds of data that are available in the field from both the field and laboratory.

WAKE A SECOND BASE after the home team's first run.

wolf-jungo-alien-ego-zentav-heslumens-llc-alterW

Wysokość poziomów od bieżącej średniej zasępu, pod którym jest określona granica

### II.3.1 Formal Representation of Nested Loop Structures

We have seen that the basic control structure used in implementing a computation is the totally nested loop. Associated with each loop in the nesting is a set of keys that it will fix and which will remain constant in the loops it contains. It is easy to see that this constraint means that the set of keys fixed within any loop is necessarily a (proper) superset of the set of keys fixed within any of its enclosing loops. Thus, the set of keys fixed within a loop is sufficient to determine its level in the nesting.

Now notice that the body of every loop (except the innermost one) contains exactly one top-level loop; thus, the body is naturally divided into three parts:

- the prolog--those actions performed before the enclosed loop
- the enclosed loop
- the epilog--those actions performed after the enclosed loop.

Conceptually, then, a totally nested loop can be represented as a list of loop descriptions, one for each of the component loops. Each such description would consist of a level identifier (indicating at which level of nesting it occurs) and the prolog and the epilog. However, during the design stage, while implementations are being developed and, in particular, when computation aggregations are being considered, it is useful to distinguish 3 classes of actions within the body of a loop:

- Prolog--those actions that *must* be performed before the enclosed loop
- Epilog--those actions that *must* be performed after the enclosed loop
- General--those actions that could end up in either the prolog or the epilog

It is also useful to separate I/O actions from the other actions. Thus, we represent each loop in the nesting as a structure of the following form:<sup>15</sup>

<sup>15</sup> This representation, and the theory of computation aggregation associated with it are due largely to the work of R. C. Fleischer [2], who improved on the earlier work of R. V. Baron.

(Level),

(Inputs<sub>P</sub>, Prolog, Outputs<sub>P</sub>)  
 (Inputs<sub>G</sub>, General, Outputs<sub>G</sub>)  
 (Inputs<sub>E</sub>, Epilog, Outputs<sub>E</sub>)

where

Level indicates the depth of the loop in the nesting

Inputs<sub>P</sub> are the files (necessarily) read in the Prolog section.

Inputs<sub>G</sub> are the files (necessarily) read in the General section.

Inputs<sub>E</sub> are the files (necessarily) read in the Epilog section.

Outputs<sub>P</sub> are the outputs generated in the Prolog section (possibly used in the enclosed loop or in the Epilog section)

Outputs<sub>G</sub> are the outputs generated in the General section.

Outputs<sub>E</sub> are the outputs generated in the Epilog section.

### II.3.2 Computation Implementation

The implementation of a computation as a nested loop structure reduces to the problem of determining how many and which levels are to be in the totally nested loop and where the I/O and computations go. The answers to these questions are constrained by the forces of necessity and efficiency.

#### II.3.2.1 Level Position of I/O and Calculations

The levels at which each input should be read, each output should be written and each calculation should be performed are determined by the following guidelines:

Inputs: Each input flow of a computation should be read at a loop level whose associated

key-tuple is identical to that of the flow's index (and on this account the totally nested loop for a computation must contain a loop corresponding to the index of each input flow). It cannot be read at a higher level because at such a level the key information is incomplete. To read it at a lower level would be inefficient, because it would cause unnecessary re-reads of the flow's records.

**Outputs:** Similarly, each output flow of a computation must be written at a loop level whose associated key-tuple is identical to that of the flow's index. It cannot be written at a higher level because of insufficient key information, and to output it at a lower level would cause multiple writes of the records.

**Calculations:** A flow expression should also be calculated at a loop level whose associated key-tuple is identical to that of the flow expression's index. Again, the key information at a higher level would be insufficient to calculate the expression, and to perform it at a lower level would be redundant. Further economy can be realized, however, in a mixed-index flow expression if it contains a sub-expression whose associated index is a sub-index of the flow expression as a whole; such a sub-expression should be split off and calculated at its appropriate (higher) level.

### II.3.2.2 Position of I/O and Calculations Within Their Assigned Levels

The placement of a read, write or calculation within a given loop level (i.e. in either the Prolog, Epilog or General section) should be done with a view toward imposing the minimum constraint on implementation. If done in this manner placement preserves the maximal flexibility in subsequent aggregation. For instance, if a calculation could go into either the Prolog or the Epilog it should be placed in the General section. If instead it were arbitrarily placed in the Epilog this unnecessary constraint would preclude subsequent aggregations that would require it to be in the Prolog (loop merging in computation aggregation is discussed below).

**Caution:** An inspection of the wind and solar array equipment depends on strength of wind and solar array equipment.

**Calculations:** A calculation must be made, *before* any work is done, *only if it must* be performed before the cutting loop is closed, *and only if it depends on something* (such as a gauge) that has *not* been calculated. The *actual* blade rate *can* be *calculated* in the *General section*.

As an obvious consequence of these guidelines it can be seen that in the case of any single-level loop or inversion loop all inputs, outputs and calculations will go into Inputs, Outputs, minimum and maximum branch logic while end of Block Transfer tokens will go into Input and the General section, respectively; Inputs, Inputs, Outputs, the Process section and the Epi log section will all be empty.

set to go to the next slide and go through the presentation again.

**H2S Enclosure** **Investigation** **Gas** **in** **Enclosure** **H2S** **in** **gas** **in** **enclosure**

**Example 1** Suppose that we have the following data about age, blood pressure, and polyp:

First, consider the PAYMENT-IN-ADVANCE problem (nothing more than getting paid) you identified in

PAY IS RATE \* HOURS IF RATE PRESENT AND HOURS PRESENT

Here, both inputs have the same index (employee-id) so there is only one loop:

Level: (employee-id)  
Inputs<sub>G</sub>: empty  
Prolog: empty  
Outputs<sub>G</sub>: empty

Inputs<sub>G</sub>: {HOURS, RATE}  
General: calculate PAY  
Outputs<sub>G</sub>: {PAY}

Inputs<sub>E</sub>: empty  
Epilog: empty  
Outputs<sub>E</sub>: empty

As explained above, everything is placed in the general sections.

Now consider a simple reduction flow equation:

ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

We have seen that the implementation of such a flow equation will always have two loop levels:

Loop 1 (outer loop)

Level: (item-id)  
Inputs<sub>G</sub>: empty  
Prolog: initialize sum  
Outputs<sub>G</sub>: empty

Inputs<sub>G</sub>: empty  
General: empty  
Outputs<sub>G</sub>: empty

Inputs<sub>E</sub>: empty  
Epilog: empty  
Outputs<sub>E</sub>: {ITEMDEMAND}

Loop 2 (inner loop)      ~~Input: DEWMID, item-id, store-id, quantity, STAR, PRICEDB, PRESENT~~**Level: (item-id, store-id)**~~Input: quantity~~      ~~Input: item-id or store-id or (binary format) symbol name of level 1 item-id or store-id~~**Prolog:**      ~~empty~~**Output:** ~~empty~~~~(b1-mixed) (b1-level)~~~~getee (getee)~~~~g1qee (g1qee)~~~~g1qee (g1qee)~~**Input:** ~~DEWMID~~**General:** calculate sum**Output:** ~~empty~~~~(STAR, PRICEDB together)~~~~STAR (STAR together)~~~~PRICEDB (PRICEDB)~~**Input:** ~~empty~~**Epi log:**      ~~empty~~**Output:** ~~empty~~~~g1qee (g1qee)~~~~g1qee (g1qee)~~

The input DEWMID has the keys item-id and store-id in its index. It must be read in the (item-id, store-id) level loop (the inner one), otherwise, computation of extended price may be performed at that level. Since this is the innermost level, everything must work out quite well.

On the other hand the output STAR and PRICEDB must be written to its index and so must be written from the outer (item-id) level loop. In order to do this, since it depends on the calculation performed in the inner loop, it must be written from the epilog of the (item-id) loop (b1-mixed) level.

**A mixed-index matching computation like:**~~g1qee (g1qee)~~~~getee (getee)~~~~g1qee (g1qee)~~**EXTENDEDPRICE IS CURRENDBER + PRICE IF ~~ITEMID~~ PRESENT****AND PRICE IS PRESENT**~~g1qee (g1qee)~~

must have two loop levels when implemented, one for each ~~ITEMID~~ of its inputs. Its representation looks like:

~~g1qee (g1qee)~~~~g1qee (g1qee)~~~~g1qee (g1qee)~~

Loop 1 (outer loop)

Level: (item-id)  
Inputs<sub>P</sub>: {PRICE}  
Prolog: empty  
Outputs<sub>P</sub> empty

Inputs<sub>G</sub>: empty  
General: empty  
Outputs<sub>G</sub> empty

Inputs<sub>E</sub>: empty  
Epilog: empty  
Outputs<sub>E</sub> empty

Loop 2 (inner loop)

Level: (item-id, store-id)  
Inputs<sub>P</sub>: empty  
Prolog: empty  
Outputs<sub>P</sub> empty

Inputs<sub>G</sub>: {CURRENTORDER}  
General: calculate EXTENDEDPRICE  
Outputs<sub>G</sub>: {EXTENDEDPRICE}

Inputs<sub>E</sub>: empty  
Epilog: empty  
Outputs<sub>E</sub> empty

### Part III: Computation Aggregation and Loop Mapping

(good status) Level 1  
(61-651) revised

As explained above the aggregation of two or more loops will result in a single totally nested loop structure. The process of computation aggregation can be performed most simply on two loops at a time (thus if it is desired to aggregate three loops, the first two are aggregated and then the result is aggregated with the third). Without loss of generality, then, we will confine the treatment that follows to pair-wise aggregation.

When two computations are found to be candidates for aggregation their suitability for aggregation must be tested, and then, if they are aggregatable, their respective totally nested loops must be merged to form a single totally nested loop and mapped to a single level. These two problems are the subjects of the next sections.

#### III.1 Loop Aggregability

IRREGULARITY (get busy)

EXTREMEDRIFT (stuck)

A little thought will show that when two nested loops are aggregated an action (read, write or calculation) in the aggregate must be performed at the same level before aggregation; there is no possibility of moving an action to a different level where it originally appeared. Thus, for two loops to be aggregatable it must be possible to construct a totally nested loop structure that contains all of the levels necessary to test. Two loops for which this is possible are said to be *level compatible* with each other.

Furthermore, there are certain ordering constraints that the actions of the individual loops satisfy and which must be satisfied by the aggregate loop: a file must be produced before it can be used; a Prolog action must occur before its associated inner loop; and an Sprolog action must occur after its inner loop.

If two computations have level compatible loops and if the ordering constraints of the two loops can be mutually satisfied in a single totally nested loop, aggregation is possible.

### III.1.1 Level Compatibility Between Loops

It is easy to show that two loops are level compatible if and only if their level structures are identical or empty levels (levels at which no actions are performed) can be inserted to make their level structures identical. Some examples of level compatible totally nested loops (TNL's) and the level structures of their aggregated results are:<sup>16</sup>

<u>loop</u>	<u>levels</u>	<u>levels in aggregate</u>
TNL <sub>1</sub>	(K), (K,L)	(K), (K,L)
TNL <sub>2</sub>	(K,L)	
TNL <sub>1</sub>	(K,L)	
TNL <sub>2</sub>	(K,L,M)	(K,L), (K,L,M)
TNL <sub>1</sub>	(K), (K,L)	
TNL <sub>2</sub>	(K,L), (K,L,M)	(K), (K,L), (K,L,M)

It is interesting to note that when aggregation occurs loop levels are neither added nor deleted; that is, the set of loop levels in the aggregate is simply the union of the sets of loop levels in the component computations.

Some examples of loops whose level structures are incompatible are:

<u>loop</u>	<u>levels</u>
TNL <sub>1</sub>	(K)
TNL <sub>2</sub>	(L)

---

<sup>16</sup> In this section the symbols K, L and M denote different keys.

TNL<sub>1</sub> (K), (K,L)  
 TNL<sub>2</sub> (L), (K,L)

TNL<sub>1</sub> (K), (K,L), (K,L,M)  
 TNL<sub>2</sub> (K), (K,M), (K,L,M)

### III.I.2 Order Constraint Compatibility Between Loops

Consider the computations for the following two flow equations:

ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

FRACTION IS DEMAND/ITEMDEMAND IF DEMAND PRESENT

It would seem immanently reasonable to aggregate these two computations since they have a common input (DEMAND) and the output of the first is an input to the second. Yet they cannot be aggregated into a totally nested loop! Their implementation descriptions reveal why. Recall that the description of the first is:

#### Loop 1 (outer loop)

Level: (item-id)

Inputs<sub>pl</sub>: empty

Prolog: initialize sum

Outputs<sub>p0</sub>: empty

Inputs<sub>sg</sub>: empty

General: empty

Outputs<sub>sg</sub>: empty

Inputs<sub>sg</sub>: empty

Epi log: empty

Outputs<sub>sg</sub>: ITEMDEMAND

Loop 2 (inner loop)

Level: (item-id, store-id)

Inputs<sub>P</sub>: empty

Prolog: empty

Outputs<sub>P</sub>: empty

Inputs<sub>G</sub>: (DEMAND)

General: calculate sum

Outputs<sub>G</sub>: empty

Inputs<sub>E</sub>: empty

Epilog: empty

Outputs<sub>E</sub>: empty

The FRACTION computation also has two nested loops:

Loop 1 (outer loop)

Level: (item-id)

Inputs<sub>P</sub>: (DEMAND)

Prolog: empty

Outputs<sub>P</sub>: empty

Inputs<sub>G</sub>: empty

General: empty

Outputs<sub>G</sub>: empty

Inputs<sub>E</sub>: empty

Epilog: empty

Outputs<sub>E</sub>: empty

Loop 2 (inner loop)

Level: (item-id, store-id)

Inputs<sub>P</sub>: empty

Prolog: empty

Outputs<sub>P</sub>: empty

Inputs<sub>G</sub>: (DEMAND)

General: do division

Outputs<sub>G</sub>: (FRACTION)

Inputs<sub>E</sub>: empty

Epilog: empty

Outputs<sub>E</sub>: empty

Clearly these computations are level compatible since they have identical level structures. But the

(Item-id) level loop of the first requires that ITEMID be an output of the (goal value) S goal  
 (bi-nest1, bi-nest1) slave

(Item-id) level loop of the second requires that it be an input to EPILOG. Their aggregate would  
 thus require records of ITEMID before they are computed, an ~~impossibly~~ impossible condition.

The basis for all ordering constraints is the simple rule that ~~an action must be produced or~~  
~~read before it is used~~. Totally nested loop implementations are designed such that this rule  
 is observed exactly. That is, things are in a loop's Prolog if and only if they ~~must~~ be done before  
 the enclosed loop(s); things are in a Epi log if and only if they ~~must~~ be performed after the  
 enclosed loop. Listed explicitly, the constraints that are ~~imposed~~ ~~when merging~~ two totally nested loops are:

- an output of an Epi log cannot be an input to a Prolog
- every Prolog action must remain in the Prolog
- every Epi log action must remain in the Epi log
- Prolog I/O must remain in the Prolog
- Epi log I/O must remain in the Epi log

Implicit are the constraints that

- an action cannot be moved from its original level to another in the (goal value) S goal  
 (bi-nest1, bi-nest1) slave
- I/O cannot be moved from its original level to another in the (goal value) S goal  
 (bi-nest1, bi-nest1) slave

The only thing that can change is that actions and their corresponding I/O can be moved  
 from the General section to either the Prolog or Epi log of the ~~original~~ ~~nesting~~ level. Such a move  
 merely reflects the addition of a constraint that does not affect ~~impossibly~~ ~~the implementation~~.

Thus, such a move may be made when necessary to ~~minimize conflicts~~ ~~minimize~~ ~~conflict~~ of the two loops  
 to be merged, but should never be done arbitrarily, so as to ~~preserv~~ ~~preserve~~ freedom in  
~~subsequent merging~~ level. Note that even some sideeffects level wise (goal value) S goal  
 (goal value) S goal

Computations whose totally nested loops are level compatible and satisfy the above order constraints are aggregatable.

### III.2 Merging Loops

Because each action and all I/O must be performed at the same level in the aggregate as it was before aggregation, the loop structure of the aggregation of two computations can be obtained through a level-by-level merge of the loop levels of the two computations to be aggregated.

The algorithm for merging two totally nested loops is:

For each loop in one:

If the other has no loop at the same level, just add the representation of that level to the description of the aggregate.

If there is a corresponding loop, the two loops must be merged into one for the aggregate.

The full details of merging loops are complicated, but a rough sketch follows. Let the corresponding loops be  $L_1$  and  $L_2$ , where no output of  $L_2$  is an input to  $L_1$ .<sup>17</sup> There are three cases:

I. Some output  $F$  of the Epilog of  $L_1$  is an input to  $L_2$ .

- a.  $F$  is an input to  $L_2$ 's Prolog section: aggregation impossible.
- b.  $F$  is used by an action in  $L_2$ 's General section: move that action to the Epilog of the corresponding level in the aggregate, along with any actions in  $L_2$ 's General section which use, as input, some output produced by the action; all other actions remain in the same sections in the aggregate as they were in  $L_1$  and  $L_2$ .
- c. All other cases: all other actions remain in the same sections in the aggregate as they were in  $L_1$  and  $L_2$ .

<sup>17</sup> Obviously, the case where no output of  $L_1$  is an input to  $L_2$  will be handled exactly the same, *mutatis mutandis*. The remain case, where each has some output that is an input to the other, is impossible.

2. Some output F, generated by some action A in the General section of L<sub>1</sub>, is an input to L<sub>2</sub>.

a. F is an input to L<sub>2</sub>'s Prolog section: move A from the General section to the Prolog section of the aggregate, along with any actions in the General section which have, as output, something used as input to that computation; all other actions remain in the same sections in the aggregate as they were in L<sub>1</sub> and L<sub>2</sub>.

b. All other cases: all actions remain in the same sections in the aggregate as they were in L<sub>1</sub> and L<sub>2</sub>.

3. Neither 1 nor 2: all actions remain in the same sections in the aggregate as they were in L<sub>1</sub> and L<sub>2</sub>.

Basically, what this means is that a General action must move to the Prolog of the aggregate if it must come before some action in that Prolog or if it must come before another General action which must be moved to the Prolog; a General action must move to the Epilog if it must come after some action in the Epilog or if it must come after another General action which must be moved to the Epilog.

### III.3 Non-Totally-Nested Loops

In this report the treatment of data driven loop implementations is restricted to loop structures that are totally nested. Totally nested implementations are not only broadly applicable, but generally simple and efficient as well. In fact they often provide the most efficient and expeditious implementations, especially when sequentially organized files, sorted by key values, are used. For the sake of completeness, though, something should be said here about non-totally-nested loops. Indeed, a great deal could be said about such implementations—enough, certainly, to make one or more separate reports. Because of this the discussion here is necessarily brief and incomplete.

Most importantly, it should be said that non-totally-nested loop structures are by no means

inefficient or uninteresting. They are used all the time and for good, solid reasons. Their use is perhaps most interesting when two or more computations cannot be performed entirely concurrently (i.e. in the same loop), but they can be performed with partial concurrency. The following two examples illustrate.

### III.3.1 Example I: Aggregating Computations with Incompatible Order Constraints

Recall the flow equations:

ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

FRACTION IS DEMAND/ITEMDEMAND IF DEMAND PRESENT  
AND ITEMDEMAND PRESENT

and their implementing computations. We saw in Section III.1.2 that the implementing computations for these flow equations could not be merged into a totally nested loop structure because the inner loop for the first had to be completed before the inner loop of the second could be performed. They can, however, be aggregated into a single loop with a structure like:

```
for each (item-id) from DEMAND
    sum = undefined
    for each (store-id) from DEMAND(item-id)
        <calculate sum>
    end
    if defined(sum) then ITEMDEMAND(item-id) = sum
    for each (store-id) from DEMAND(item-id)
        <calculate FRACTION>
    end
end
```

This is a non-totally nested loop structure, since two loops (the inner ones) appear at the same level.

It is interesting to compare this aggregate implementation with the unaggregated implementation of the two computations involved (as separate loops in separate job steps). On the one hand, in either implementation every record of the DEMAND flow must be accessed twice, so no accesses are eliminated by aggregation. On the other hand, accesses of the records of the ITEMDEMAND flow are eliminated by aggregation. If the computations are implemented separately, every record of ITEMDEMAND must be written into a file by the first computation and then read back by the second; whereas in the aggregate implementation the records are used as they are generated, so no re-reading is necessary.<sup>18</sup>

In general we have seen that when two implementations are level-compatible, the only case in which their aggregate cannot be implemented as a totally nested loop is where, for some loop level, the output of the Epi:og section of one is an input to the Pro:og section of the other (as is the case with ITEMDEMAND above). In such a case the corresponding loop level of the aggregate can be implemented (as above) as two loops of the same level performed in sequence, and re-reads of the flow in question will be saved.

### III.3.2 Example 2: Aggregating Computations That Are Not Level-Compatible

In Section III.1.1 we saw that computations with the following level structures were not level compatible with one another:

TNL <sub>1</sub>	(K), (K,L), (K,L,M)
TNL <sub>2</sub>	(K), (K,M), (K,L,M)

The fact that they are not level-compatible means that it is impossible to devise a total

---

<sup>18</sup> In fact, if these records are not used by any other computation in the data processing system, it is not necessary to write them out into a file either.

nesting of loops that will implement their aggregate. They might, however, be said to be *partially* level-compatible, since the outermost levels have identical keys. If a common driver set can be found for that level, they might be implemented as a non-totally-nested loop structure. The following is a possible implementation skeleton:

```
for each (K) from D0
    for each (L) from D1
        for each (M) from D2
            .
            .
            end
    end

    for each (M) from D3
        for each (L) from D4
            .
            .
            end
    end

end
```

where the  $D_i$  are distinct drivers.

This is another commonly found construct in file data processing. It is the case where, for a common set of values for the sub-index (K), two or more independent computations are to be performed. As in the previous example, there is some I/O saving (over separate implementations of the computations involved) because each record of  $D_0$  has to be read only once.

efficiency and for better code generation, we have to implement the code based on index sets.

We have seen that, by defining index sets containing the union of all or some of the driving flow sets, we derive it. These flows are used to determine which index set to use for the loop index. The body of the loop is performed once for each loop iteration. This is a significant improvement.

We have also seen that, in general, computations and data flows will often be implemented by nested loop structures. That is, an implementation involves one or more loops, each of which must have a driving flow set.

In Part I we saw that for a computation as a whole ~~critical index sets~~<sup>bns</sup> requires the effective enumeration of the critical index sets of each of its loops. This constraint obviously extends to the individual loop levels. Additionally, the ~~critical driving loops~~<sup>bns</sup> of a level must be effectively enumerated so that all repeats will be visited. Considering these constraints in terms of drivers we have

### The Fundamental Data Driven Loop Driver Constraints

In the nested loop structure implementing a data driven computation, the drivers for each level, i, must enumerate no index set  $I_i$  such that

is not covered by at least one of the driving sets of the innermost loop iteration.

1. for every input flow  $F_i$  at level i,  $I_{i+1}$  is included in at least one output flow  $F_j$  at level j, such that  $I_j$  is included in at least one of the driving sets of the innermost loop iteration.

2. for every output  $F_j$  at level i,

some flow based on  $I_i$  is present in the outermost iteration (innermost iteration) of  $F_j$ .

In order to discuss the determination of loop level drivers we must first develop a precise theory of index sets and critical index sets.

#### IV.1 A Theory of Index Sets and Critical Index Sets for Data Driven Loops

Let us begin with some definitions and useful consequences of these definitions.

##### IV.1.1 Definitions and Useful Lemmas

We redefine the notions of a flow's index set and critical index set formally and introduce the operators Proj, Inj and Restr:

Definition: The *index set* of a flow F with index I is defined as

$$IS(F) = \{I \mid \text{there is a record in } F \text{ for } I\}$$

Definition: The *critical index set* of a flow F (with index I) with respect to a flow X is defined as

$$CIS_X(F) = \{I \mid \text{there is a record in } F \text{ for } I \\ \text{that is necessary to generate some record in } X\}$$

Definition: The *projection* of an index set S with index  $(k_1, \dots, k_m, k_{m+1}, \dots, k_n)$  onto the sub-index  $(k_1, \dots, k_m)$  is defined as

$$\text{Proj}(S, (k_1, \dots, k_m)) = \\ \{(k_1, \dots, k_m) \mid \exists (k_{m+1}, \dots, k_n) \text{ such that } (k_1, \dots, k_m, k_{m+1}, \dots, k_n) \in S\}$$

Definition: The *injection* of an index set S with index  $(k_1, \dots, k_m)$  by the index set T with super-index  $(k_1, \dots, k_m, k_{m+1}, \dots, k_n)$  is defined as

$$\text{Inj}(S, T) = \\ \{(k_1, \dots, k_m, k_{m+1}, \dots, k_n) \mid (k_1, \dots, k_m) \in S \wedge \\ (k_1, \dots, k_m, k_{m+1}, \dots, k_n) \in T\}$$

Definition: The *restriction* of an index set S with index  $(k_1, \dots, k_n)$  by the condition C (whose truth depends on the values of the keys  $k_1, \dots, k_n$ ) is defined as

$$\text{Restr}(S, C) = \{(k_1, \dots, k_n) \in S \mid C \text{ is true}\}$$

From the last three definitions the following simple but useful results (stated without proof) can be obtained:

**Lemma 1:** If A is an index set with index I, then

## Index Function

**Proj(A,B)** = the subset of  $\{0, 1\}^A$  whose entries in  $B$  have value 1  $\rightarrow$  proj(A,B) = A \ B

**Lemma 2:** If  $A$  and  $B$  are index sets with the same index, then  $\text{Inj}(A,B) = B \cap \text{Inj}(A,A)$

$$\text{Inj}(A,B) = B \cap \text{Inj}(A,A)$$

defining inj(A,B) = {f : f is a function from A to B such that f is one-to-one}

In particular, if  $A$  and  $B$  are index sets with the same index, then

all functions from  $A$  to  $B$  will map the subset  $\{0, 1\}^A$  to the subset  $\{0, 1\}^B$  & to another set smaller than  $W$

$$\text{Inj}(A,B) = B \cap A$$

defining inj(A,A) = {f : f is a function from A to A such that f is one-to-one}

**Lemma 3:** If  $S$  and  $T$  are index sets where the index of  $T$  is a super-index of that of  $S$ , then

as function of  $X$  will be to  $T$  what  $S$  is to  $S$

$$\text{Inj}(S,T) \subseteq T$$

$$\{f : f \in \text{Inj}(S,T) \text{ & } f(S) \subseteq T\} = (\exists)2^1$$

**Lemma 4:** If  $T$  is an index set with index  $I_T$  and  $S$  is an index set with index  $I_S$ , a sub-index of  $I_T$ , then

$$\{f : f \in \text{Inj}(S,T) \text{ & } f(S) \subseteq T\} = (\exists)2^{I_S}$$

$$\text{Proj}(\text{Inj}(S,T), \text{Inj}(S,T)) = \text{Inj}(S, \text{Inj}(S,T))$$

definition  $I_S$  is an index set with the same size as  $S$  but with super-index of  $T$  as function of  $T$

of  $I_S$ , then as function of  $I_S$  will be to  $T$  what  $S$  is to  $T$

$$\text{Restr}(S, F \text{ PRESENT}) = \text{Inj}(IS(F), S) = \{(x_1, \dots, x_n) \in S \mid \forall i \in I_S \exists j \in I_T \text{ s.t. } (x_i, j) \in F\}$$

$$(x_1, \dots, x_n) \in S \text{ & } \forall i \in I_S \exists j \in I_T \text{ s.t. } (x_i, j) \in F \in \{(x_1, \dots, x_n)\}$$

**IV.12 Critical Index Sets Elimination by Computation**  $\rightarrow$  2 for xobel is to nothing in  $T$

We begin with two theorems concerning the critical index sets of flows involved in computations. The results are expressed in terms of the index sets of the inputs and outputs.

$\wedge 2 \in \{x_1, \dots, x_n\} \setminus \{x_{i_1}, \dots, x_{i_m}\} \wedge \{x_{i_1}, \dots, x_{i_m}\} \subseteq \{x_1, \dots, x_n\}$

**Theorem 1:** If  $F$  is a flow defined in terms of the flows  $F_1, \dots, F_n$  by a non-reduction flow equation, where each flow  $F_i$  has index  $I$   $\rightarrow$  2 for xobel is to nothing in  $T$

$$CIS_F(F_i) = \text{Proj}(\text{IS}(F_i), \text{Inj}(S, \{x_1, \dots, x_n\})) \text{ s.t. } \text{is to nothing in } T$$

That is, an input record is needed in the evaluation of a flow  $F$  if and only if it depends on index or sub-index of  $x_i$   $\rightarrow$  nothing in  $T$  is dependent on  $x_i$ . By Lemma 1 we have that

nothing in  $T$  is dependent on  $x_i$

nothing in  $T$  is dependent on  $x_i$  as a 1 if it is not

**Corollary 1:** Let  $F$  be defined as in Theorem 1. Then for any flow  $F_i$  with index identical to that of  $F$

$$\text{CIS}_F(F_i) = \text{IS}(F)$$

**Theorem 2:** If  $R$  is a flow (with index  $I_R$ ) described by the application of a reduction operator to a flow expression  $\text{expr}$  in terms of the flows  $F_1, \dots, F_n$ , where each flow  $F_i$  has index  $I_i$  (e.g. the flow equation for  $R$  is:  $R \text{ IS SUM OF } \text{expr} \text{ FOR EACH } \langle I_R \rangle$ ), then

$$\text{CIS}_R(F_i) = \text{Proj}(\text{IS}(\text{expr}), I_i)$$

(Note that the index of  $\text{expr}$  must be a super-index of  $I_R$ .)

This theorem simply says that when a flow (as that described by  $\text{expr}$ ) is reduced every record of that flow is used in calculating the result. From Theorem 1 we have in turn that the critical index set of each  $F_i$  with respect to the flow to be reduced is given by the expression on the right-hand side of the above equation.

**Corollary 2:** If  $R$  is a flow (with index  $I_R$ ) described by the application of a reduction operator to a flow  $F$  (e.g.  $R \text{ IS SUM OF } F \text{ FOR EACH } \langle I_R \rangle$ ), then

$$\text{CIS}_R(F) = \text{IS}(F)$$

The following theorems concern the nature of the index sets of flow expressions. First, a simple result about flows described by reduction:

**Theorem 3:** If  $R$  is a flow (with index  $I_R$ ) described by the application of a reduction operator to a flow expression  $\text{expr}$  (e.g. the flow equation for  $R$  is:  $R \text{ IS SUM OF } \text{expr} \text{ FOR EACH } \langle I_R \rangle$ ), then

$$\text{IS}(R) = \text{Proj}(\text{IS}(\text{expr}), I_R)$$

This theorem says that there will be a valid arithmetic expression whose value is the flow to be reduced has at least one corresponding record.

(3.21 = 3.21)

$$(3.21 = 3.21)$$

For flows described by non-reduction flow expressions a more extensive treatment is necessary. We begin with simple arithmetic expressions and then extend them to conditional expressions.

In FE-HIBOL every arithmetic flow expression reduces to a flow expression involving well sets of the flows involved in it (it can, of course, be qualified similarly). We then make an expression a simple arithmetic flow expression (safe) and do its reduction (safe) to reduce it to a simple arithmetic expression (safe) and do its reduction (safe) to reduce it to a simple arithmetic expression (safe).

To model safe arithmetic expression (3.21 = 3.21) we must have a flow expression involving well sets of the flows involved in it (it can, of course, be qualified similarly). We then make an expression a simple arithmetic flow expression (safe) and do its reduction (safe) to reduce it to a simple arithmetic expression (safe).

**AND F<sub>i</sub> PRESENT**

where  $F_i$  is a flow having the same index as  $F_j$ . If  $F_i$  is not present in  $F_j$ , then  $F_j$  is not present in  $F_i$ . If  $F_i$  is present in  $F_j$ , then  $F_j$  is present in  $F_i$ . If  $F_i$  is not present in  $F_j$ , then  $F_j$  is not present in  $F_i$ .

**Theorem 4:** The index safety expression  $(3.21 = 3.21)$  is safe if and only if  $\exists F_1 \dots F_n$  where  $F_i$  is a flow having the same index as  $F_j$  (i.e.,  $i = j$ ) and  $\forall i \in \{1, 2, \dots, n\}$  there must be at least one such flow.

$$(3.21 = 3.21)$$

**IF n = 1**  
 $\exists F_1$  such that  $F_1$  is a flow having the same index as  $F_j$  (i.e.,  $i = j$ ) and  $\forall i \in \{1, 2, \dots, n\}$  there must be at least one such flow.

**IF n > 1**  
 $\exists F_1, \dots, F_n$  such that  $F_i$  is a flow having the same index as  $F_j$  (i.e.,  $i = j$ ) and  $\forall i \in \{1, 2, \dots, n\}$  there must be at least one such flow.

**Corollary 3:** Let safe  $F_1, \dots, F_n$  be a flow expression defined as in Theorem 4 with the additional constraint that the  $F_i$  are of uniform index. Then

$$(3.21 = 3.21)$$

$$\begin{aligned} \text{IS(safe}[F_1, \dots, F_n]) = & \quad \text{IS}(F_1 \text{ if } n \leq 1) \\ & \cap \text{IS}(F_i \text{ if } n > 1) \end{aligned}$$

As mentioned above the only legal arithmetic flow expression in FE-HIBOL is a safe or a safe further qualified by some condition. This further qualification must take the form of a logical expression ANDed with the safe. Thus, to complete our treatment of arithmetic flow expression we only need the following simple theorem:

**Theorem 5:** The index set of a simple arithmetic flow expression safe qualified by the condition C is given by

$$\text{IS(safe AND C)} = \text{Restr}(\text{IS(safe)}, C)$$

Consideration of special cases leads to three simple corollaries:

**Corollary 4:** By Lemmas 2 and 5

$$\begin{aligned} \text{IS(safe AND G PRESENT)} &= \text{Inj}(G, \text{IS(safe)}) \\ &= \text{IS(safe)} \cap \text{Inj}(G, \text{IS(safe)}) \end{aligned}$$

**Corollary 5:**

$$\text{IS(safe AND (C}_1 \text{ AND C}_2\text{))} = \text{Restr}(\text{IS(safe)}, C_1) \cap \text{Restr}(\text{IS(safe)}, C_2)$$

**Corollary 6:**

$$\text{IS(safe AND (C}_1 \text{ OR C}_2\text{))} = \text{Restr}(\text{IS(safe)}, C_1) \cup \text{Restr}(\text{IS(safe)}, C_2)$$

For conditional expressions with two cases<sup>19</sup> we have the following result:

**Theorem 6:** Let E be a conditional flow expression of two terms:

$$\begin{aligned} E = & \quad \text{expr}_1 \text{ IF } C_1 \\ \text{ELSE } & \text{expr}_2 \text{ IF } C_2 \end{aligned}$$

---

<sup>19</sup> The extension of this theorem to more than two cases is trivial.

where  $\text{expr}_1$  and  $\text{expr}_2$  are legal PEGMOL flow expressions and  $C_1$  and  $C_2$  are logical expressions. Define the flow expression  $E_1$  and  $E_2$  (using the same flow and logical expressions):

$E_1 = \text{if } \text{expr}_1 \text{ then } \text{IF } C_1 \text{ else } \text{expr}_1$  (no set avoids benefit from  $C_1$ )

$E_2 = \text{if } \text{expr}_2 \text{ then } \text{IF } C_2 \text{ else } \text{expr}_2$  (no set avoids benefit from  $C_2$ )

Then  $E = E_1 \vee E_2$  (no set avoids benefit from either  $C_1$  or  $C_2$ )

$$\text{ISE}(E) = \text{ISE}(E_1) \vee \text{ISE}(E_2)$$

(monotonicity of well-formed sets from  $\text{IF}$ )

IV.3 Examples

To illustrate the above theorems we give a few examples of loop implementations and verify that the loop level drivers satisfy the fundamental driving conditions.

Example E R IS THE SUM OF F FOR EACH k<sub>1</sub>

where R has index  $(k_1)$  and F has index  $(k_1, k_2)$ . As we have seen above the typical implementation is:

$$(0, \text{else})21 \text{ fresh } R, 0, \text{else}21 \text{ fresh } F = (0, 0 \text{ MA else})21$$

$$(\text{else}21, 0)(n1, 0) (\text{else}21 =$$

for each  $(k_1)$  from F

$$(0, \text{else}21) \text{ fresh } R, 0, \text{else}21 \text{ fresh } F = (0, 0 \text{ MA }, 0) \text{ MA else}21$$

for each  $(k_2)$  from F( $k_1$ )

$$(0, \text{else}21) \text{ fresh } R, 0, \text{else}21 \text{ fresh } F = (0, 0 \text{ MA }, 0) \text{ MA else}21$$

else = ...

end (then group of sets avoid  $\text{else}$  first so draw knowlesque for T)

if defined then begin

$$R(k_1) = \text{sum } 0.71 \text{ max } 0.71 \\ \text{write } R(k_1) \text{ IF } \text{ sum } < 32.13 \\ \text{end}$$

end

end (group of sets and start of method  $\text{void}$  to no longer see T)

In level 1 we have the output R and the driver F. The index set  $D_1$  enumerated by this driver at this level is<sup>20</sup>

$$D_1 = \text{Proj}(\text{IS}(F), (k_1)) = \text{IS}(R) \quad (\text{by Theorem 3})$$

thus satisfying the driving constraint for the input R.

In level 2 we have the input F and the driver F. The index set  $D_2$  enumerated by this driver at this level is

$$D_2 = \text{IS}(F) = \text{CIS}_R(F) \quad (\text{by Corollary 2})$$

thus satisfying the driving constraint for the output F.

Example 2:

PAY IS      HOURS \* 3.00      IF HOURS PRESENT AND  
NOT HOURS > 40

ELSE 120 + (HOURS - 40) \* 4.5    IF HOURS PRESENT

We shall use this example to illustrate Theorem 6. Define  $E_1$  and  $E_2$  by

$E_1 = \text{HOURS} * 3.00 \quad \text{IF HOURS PRESENT AND NOT HOURS} > 40$   
and

$E_2 = 120 + (\text{HOURS} - 40) * 4.5 \quad \text{IF HOURS PRESENT AND}$   
 $\text{NOT (HOURS PRESENT)}$   
 $\text{AND NOT HOURS} > 40)$

By pure logical simplification the last equation can be rewritten:

$E_2 = 120 + (\text{HOURS} - 40) * 4.5 \quad \text{IF HOURS PRESENT AND}$   
 $\text{HOURS} > 40$

From Theorem 6 we have that

---

<sup>20</sup> Theorem 8 of the next section provides a formal treatment of enumerated index sets.

- Restr(IIS00005),  $\text{NET\_MEMS} > 400$  (by Theorem 5) CS at level 2(1)
  - U Restr(IIS00005),  $\text{NET\_MEMS} > 400$  (by Theorem 5)
 
$$(\text{CS}(T \text{ ref}) - (\text{CS}(T) + \text{CS}(M(TD2))) \text{ (ref)} = 0)$$
  - Restr(IIS00005),  $\text{NET\_MEMS} > 400 \wedge \text{NET\_MEMS} > 400$ 

A jusqu'à présent nous n'avons pas fait de chose qui faites sens
  - Restr(IIS00005), T
  - IIS00005

and by Corollary 1

**CIS<sub>new</sub>(MURSI) = IS(PAY) - IS(URSIS)** (Eq. 15) = CIS<sub>old</sub>(F) - CIS<sub>new</sub>(F)

### **Example 3**

where EP and D have the domain  $\{0, \dots, n-1\}$ , where  $i \in D$  and P has the index  $\{0, \dots, m-1\}$ . (This is our familiar EXTENDED operation  $\rightarrow$  CONTRACTED operation). CURRENTLY abbreviated by EP, P and C, and the  $\delta$ -operator  $\delta$  denotes the difference operator  $\delta = \text{EP} - \text{C}$ .

We have that  $g_A \in \text{gr}(\mathcal{O}_H)$  for  $\mathcal{O}_H = \text{TM}(M, \mathbb{R})$ .

**CIS-101 - ISOMERS PRESENT IN CHLOROETHANE - 2004 + 150 = 350**

**CIS<sub>0</sub>(IP)** - Proj(IIS4EP), (1st-100)  
- Proj(IIS4EP) (1st-100)  
- ISWP & Proj(USID), (1st-100)  
By parts logic! 2nd half  
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19 Q20 Q21 Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32 Q33 Q34 Q35 Q36 Q37 Q38 Q39 Q40 Q41 Q42 Q43 Q44 Q45 Q46 Q47 Q48 Q49 Q50 Q51 Q52 Q53 Q54 Q55 Q56 Q57 Q58 Q59 Q60 Q61 Q62 Q63 Q64 Q65 Q66 Q67 Q68 Q69 Q70 Q71 Q72 Q73 Q74 Q75 Q76 Q77 Q78 Q79 Q80 Q81 Q82 Q83 Q84 Q85 Q86 Q87 Q88 Q89 Q90 Q91 Q92 Q93 Q94 Q95 Q96 Q97 Q98 Q99 Q100

Smart drivers by C

```

for each (item-id) from C
    get P(item-id)
    for each (store-id) from C(item-id)
        get C(item-id, store-id)
        EP(item-id, store-id) = ...
        if defined[EP(item-id, store-id)]
            then write EP(item-id, store-id)
    end
end

```

In level 1 the input is P and the driver is C. The index set  $D_1$  enumerated by this driver at this level is

$$D_1 = \text{Proj}(\text{IS}(C), (\text{item-id})) \\ \geq \text{IS}(P) \cap \text{Proj}(\text{IS}(C), (\text{item-id})) = \text{CIS}_{\text{EP}}(P)$$

In level 2 the input is C, the output is EP and the driver is C. The index set  $D_2$  enumerated by this driver at this level is

$$D_2 = \text{IS}(C) \\ \geq \text{Inj}(\text{IS}(P), \text{IS}(C)) \quad (\text{by Lemma 3}) \\ = \text{CIS}_{\text{EP}}(C) = \text{IS}(EP)$$

Thus we see that the flow C is (at least) adequate to drive both levels.

#### IV.1.4 Driving Flow Set Sufficiency

We wish to be able to determine whether a set of input flows is sufficient to drive a computation loop level. Let us begin by defining the notion of the necessary index set for a computation level:

**Definition:** The necessary index set at level i for a computation C (denoted  $\text{NIS}_i(C)$ ) is defined as the set of index values necessary to drive level i of the totally nested loop implementing C.

By the fundamental driving constraint we have

$\exists \text{ const } (\text{bi-est})$  does not

**Theorem 7:** The necessary index set for level 1 of a computation  $C \in \text{IS}(F)$  is

$$\text{NIS}_1(C) = \left( \bigcup_{\substack{F \in \text{IS}(C), \\ I \in \text{I}_1(C)}} \text{IS}_1(I) \right) \setminus \{ \text{bi-est} \} \quad \begin{array}{l} \text{bi-est} \text{ does not} \\ \text{const} \end{array}$$

$\in \text{IS}(C)$

$\text{bi-est} \in \text{IS}(C)$

$\text{bi-est} \in \text{IS}(C)$

$\dots = \{\text{bi-est}\}$

where  $\text{O}(C)$  = outputs of computation  $C$

$\text{O}_1(C)$  = outputs of level 1  $\{\text{bi-est}, \text{bi-est}\}$  (see Table 11)

$I_1(C)$  = inputs of level 1  $\{\text{bi-est}, \text{bi-est}\}$  at this level

Now a loop level can be driven by inputs only at the same or lower levels (those at higher levels do not have enough keys in their indices). Obviously the index set enumerated by a driving input at the same level is its index set. The index set enumerated at a level by a driving input at a lower level is given by the following theorem:

**Theorem 8:** The index set  $S_F(I)$ , generated by a driving input  $F$  at level  $I$  by an input  $F$  read

at a lower level is  $S_F(I) = \{ \text{bi-est} \}$  if and only if  $F$  is a driving input at level  $I$ .

$$S_F(I) = \text{Proj}(\text{IS}(F), I)$$

at level  $I$  is  $\{ \text{bi-est} \}$

$$(I)21 = \emptyset$$

Using the terminology just introduced we have

$$(I)21, (I)21 \neq \emptyset$$

$$(I)21 = (I)21 \neq \emptyset$$

**Theorem 9:** A set  $B$  of flows is sufficient to drive level 1 (with index 1) if and only if

$$\text{NIS}_1(C) \subset \bigcup_{F_j \in B} S_{F_j}(1)$$

that is, if and only if the index set enumerated by  $B$  at level 1 contains the necessary index set

for that level.

is not the  $\text{xbn}$  you can see in the bottom right quadrant of the diagram.

level contributions

<sup>21</sup> There is some redundancy in this expression. The critical index set of any level  $I$  is defined with respect to  $\{ F \in \text{IS}(F) \mid \text{IS}(F) \cap \text{IS}(C) \neq \emptyset \}$ . This is true by Corollary 4 since the fact that the input and output must have identical indices. Then  $\text{IS}(F) \cap \text{IS}(C) = \text{IS}(F \cap C)$ . So the union  $\{ F \in \text{IS}(F) \mid \text{IS}(F) \cap \text{IS}(C) \neq \emptyset \}$  would suffice to perform the first union over just those  $F \in \text{IS}(C) - \text{IS}(C)$ .

#### IV.1.5 Minimal Driving Flow Sets

The set of all inputs of a computation is sufficient to drive that computation. We are interested in finding the smallest subsets of this set that will provide sufficient drivers for each level. This interest stems from our implementation constraint that all drivers must be read sequentially and must have compatible sort orders. If all contained inputs were used to drive each level of a computation loop, all inputs to that computation would have to have compatible sort orders and all would have to be read sequentially, a constraint that is often unnecessarily severe.

Moreover, from an efficiency point of view, we generally want the set of indices enumerated by the drivers at any level to be as small as possible (while satisfying the fundamental driving constraints) so as to minimize the number of iterations. For example, if we are trying to minimize I/O accesses and we have a loop that reads some (non-driving) flow by random access, the fewer iterations there are the fewer attempts there will be to access records from that flow.

Consider, for example, the EP computation (Example 3 above). The inputs contained in the outer loop are P and C. Both together could have been used as a driving flow set for that level. We were able to show, however, that C alone was sufficient to drive the outer loop. Thus, we came up with an implementation in which only the flow C had to be sorted and read sequentially. Additionally, in this implementation only those records of P that can actually be used are fetched.

It is important to note that the using some smallest driving flow set for each level does not always improve efficiency. In the computation above it can be shown that P alone is sufficient to drive the outer loop. However, such an implementation would be no better than one in which the outer loop is driven by both inputs. Since the inner loop must be driven by C in any case, we would still end up using both inputs as drivers; both would have to be sorted compatibly and read sequentially; and more records of P would be read than would actually be used.

## **¶ 2. Determination of Index-Set Induction**

To show that a flow set is a legal driving flow set we must prove the set inclusion  $\text{flow set } \text{NIS}_1 \text{ mobnet } \neq \emptyset \text{ wolt } (\text{yavivib-nan}) \text{ since about half of all } \text{NIS}_1 \text{ sets are } \text{NIS}_1 \text{ mobnet. The relationship } \text{NIS}_1 \text{ is a } \text{NIS}_1 \text{ given above. In an } \text{NIS}_1 \text{ way, we have in fact already shown some wolt half mob net subsets of all flow result elements wolt and the right mobnet. such inclusions for some of the examples given above. There is a simple formal method of } \text{NIS}_1 \text{ in both cases given in T. (works for signals). } \text{NIS}_1 \text{ set elements for mobility by reviewing set inclusions}^{29} \text{ based on the use of the concept of flow.}$

The characteristic function of an index set is a logical expression (or predicate) whose variables are the keys of the index of that index set. For a given index value, it evaluates to true (true) if and only if the index set contains that value otherwise it evaluates to false. Our search technique rests on the equivalence of set inclusion with an implication ( $\rightarrow$ ) relationship between the characteristic functions of the two sets involved. In particular, for sets A and B with characteristic functions  $A_{char}$  and  $B_{char}$  respectively,

<sup>22</sup> Not every minimal driving flow set may be feasible. In Proposition 1, the least-cost-flow policy employs this requirement, given that the flows in the solution must be non-negative. These can be sorted in an order consistent with the resulting value of the objective function.

<sup>22</sup> Insofar as set inclusion is provable. It can be shown that the general question of provability of inclusion is not solvable (see *ed. 1* section below) and has not been so shown (see *ed. 2* section below).

$$A \supset B \leftrightarrow B_{\text{char}} \rightarrow A_{\text{char}}$$

The expression on the right of the equivalence symbol ( $\leftrightarrow$ ) is a formula in the first order predicate calculus. If this formula can be shown to be a tautology the corresponding set inclusion is proved. Showing that a formula is a tautology is equivalent to showing that it simplifies to T. Since powerful first order predicate calculus simplifiers exist, the task of proving set inclusion can be solved by recasting the hypothesis as a predicate calculus formula and trying to simplify it. If it can be simplified to T inclusion is proved; if it simplifies to F inclusion is disproved.

When the formula cannot be simplified to either T or F, the meaning of the result is not clear. Either the simplification is correct (in which case the formula is not a tautology, and thus set inclusion does not hold) or the simplifier has run up against a fundamental limitation<sup>24</sup> and has failed to simplify the formula completely. In the latter case the formula may in fact be equivalent to T (implying set inclusion), but the simplifier is unable to determine it. Because of this ambiguity, the wisest assumption is the conservative one: whenever simplification to T does not occur, set inclusion does not hold.

#### IV.2.1 Characteristic Functions for Index Sets

In this section the particulars of the syntax<sup>25</sup> and semantics of characteristic functions for index sets are presented.

The characteristic function for an index set is a logical expression (predicate) in terms of its keys of its index that is true for an assignment of values to those keys in exactly those cases in

---

<sup>24</sup> It is a well-known fact that it is impossible to devise a procedure that will correctly simplify every formula in the first order predicate calculus.

<sup>25</sup> Because our work is implemented in the LISP programming language the notation is unabashedly LISPish.

which the index set contains a corresponding index value. That is, if  $S_{char}(k_1, \dots, k_n)$  denotes the characteristic function for the index set  $S$  then

$$S_{char}(k_1, \dots, k_n) = T \text{ iff } S \text{ contains an index value with } k_1 = k_1, \dots, k_n = k_n$$

The logical operators from which characteristic functions are formed are:

### 1. Standard logical operators<sup>26</sup>

- a. AND (AND p<sub>1</sub>, ..., p<sub>n</sub>) = T for a particular key-tuple instance iff all of the p<sub>i</sub> are true for that instance
- b. OR (OR p<sub>1</sub>, ..., p<sub>n</sub>) = T for a particular key-tuple instance iff any of the p<sub>i</sub> are true for that instance
- c. NOT (NOT p) = T for a particular key-tuple instance iff p is false for that instance
- d. FOR-SOME (FOR-SOME (k<sub>1</sub>, ..., k<sub>n</sub>) p(k<sub>1</sub>, ..., k<sub>m</sub>, k<sub>m+1</sub>, ..., k<sub>n</sub>)) = T for a particular key-tuple instance (k<sub>m+1</sub>, ..., k<sub>n</sub>) iff there exist values for the keys k<sub>1</sub>, ..., k<sub>m</sub> such that the predicate p(k<sub>1</sub>, ..., k<sub>n</sub>) is true; this is existential quantification.

### 2. Standard arithmetic comparison operators (their arguments must be arithmetic expressions in terms of variables (see below) and constants formed using the arithmetic operators +, -, \* and /)

- a. EQUAL (EQUAL expr<sub>1</sub>, expr<sub>2</sub>) = T iff expr<sub>1</sub> and expr<sub>2</sub> have the same numerical value
- b. GREATERP (GREATERP expr<sub>1</sub>, expr<sub>2</sub>) = T iff the numerical value of expr<sub>1</sub> is greater than that of expr<sub>2</sub>
- 3. The special operator DEFINED: (DEFINED (V per (k<sub>1</sub>, ..., k<sub>n</sub>))) = T iff there is a record in the variable V in period per for the key-tuple instance (k<sub>1</sub>, ..., k<sub>n</sub>). The argument to a DEFINED operator must be a variable.

The terms introduced here are explained in greater detail in the following sections.

---

<sup>26</sup> The symbols p and p<sub>i</sub> denote predicates.

#### IV.2.1.1 Variables

A *variable* is a representation of a HIBOL flow with key and period information attached. The period uniquely identifies the variable in time (i.e. it specifies a particular "incarnation" of the flow). An assignment of values to a variable's index and its period specifies an *instance* of that variable and this instance is said to be *defined* if there is a datum (and thus record) corresponding to the key and period values named in the assignment.

The general form for a variable is

(flow-name period key<sub>1</sub> ... key<sub>n</sub>)

where **flow-name** is the name of the associated flow<sup>27</sup>, the slot **period** contains the name of the period in which the variable is generated or input, and the slots **key<sub>i</sub>** contain the names of the keys of the variable. An example of a variable specification is

(ENROLLED term student subject-number)

where

ENROLLED is the name of the variable

term is the name of a period

student and subject-number are the names of the variable's keys

An occurrence of a variable in a predicate is called a *variable reference*. In a variable reference the form in the period slot identifies a particular incarnation of the variable (e.g. if the period slot contains TERM that means that this term's incarnation of the variable is being referred to; if it contains (PLUS TERM -1.), last term's incarnation is referred to).

---

<sup>27</sup> The variable and the flow have the same name.

IV.2.1.2 (DEFINED variable-reference)

This expression is true if and only if variable-reference is defined. In particular an expression like

(DEFINED (ENROLLED term student subject-number))

is true for an assignment of constant values to each of its keys and its period if and only if the variable ENROLLED in the specified period contains a record corresponding to the specified index value; otherwise it is false. Thus, for example, the predicate above is true for subject-number = 33 and term = TERM if and only if in this term's incarnation of ENROLLED there is a record for the index value 4(JOE 33) (i.e. if and only if Joe is enrolled in subject #33 during the current term).

IV.2.1.3 Correspondence Between Logical and Set Theoretic Notations

In our characteristic function/index set duality the general correspondence between logical and set operators is given by:

<u>logical operator</u>	<u>set operator</u>
AND	$\cap$
OR	$\cup$
(FOR-SOME ( $k_1, \dots, k_n$ ) $S_{char}$ )	$\text{Proj}(S, (k_1, \dots, k_n))$
(AND $S_{char}$ C)	$\text{Restr}(S, C)$
(AND $S_{char}$ T <sub>char</sub> )	$\text{Inj}(S, T)$
(DEFINED (V ...))	$\text{IS}(V)$

That is:

the characteristic function of the intersection of two sets is the logical AND of their characteristic functions;

the characteristic function of the union of two sets is the logical OR of their characteristic functions;

the characteristic function of the projection  $\text{Proj}(S, I')$  of an index set S onto the sub-index  $I'$  is the FOR-SOME operator applied to the characteristic function of S and the remaining keys;

the characteristic function of the restriction  $\text{Restr}(S, C)$  of an index set  $S$  by the condition  $C$  is the logical AND of the characteristic function of  $S$  and the condition  $C$ ;

the characteristic function of the injection  $\text{Inj}(S, T)$  of an index set  $S$  by the index set  $T$  is the logical AND of their characteristic functions;

the characteristic function of the index set  $\text{IS}(V)$  of a variable  $V$  is the DEFINED operator applied to that variable.

This mapping can be used to determine the characteristic function of any set expression encountered above.

#### Examples:

The index set

$\text{IS}(P)$

has the characteristic function

(DEFINED (P DAY item-id))

The index set

$\text{IS}(P) \cap \text{Proj}(\text{IS}(C), (\text{item-id}))$

has the characteristic function

(AND (DEFINED (P DAY item-id))  
(FOR-SOME (store-id) (DEFINED (C DAY item-id store-id))))

The index set

$\text{Restr}(\text{IS(HOURS)}, \text{NOT HOURS} > 40)$

has the characteristic function

(AND (DEFINED (HOURS WEEK employee-id))  
(NOT (GREATERP (HOURS WEEK employee-id) 40))))

#### IV.2.2 Back-Substitution of Characteristic Functions

We would like our characteristic functions to contain as much information as possible so as to be able to determine as much as possible about the inclusion properties of index sets.

The only possible characteristic function for a variable ( $V$  per  $(k_1, \dots, k_n)$ ) that is a system input (i.e. a variable whose flow is *not computed* by the system; for example a supplier list) is the trivial one (DEFINED ( $V$  per  $(k_1, \dots, k_n)$ )), because all that can be said is that it contains a record iff it contains a record.

In some cases an input variable may have the special property that it will always contain a record for every allowable index value. (Knowledge of such a property cannot be deduced from the HIBOL specification of a data processing system; it must be supplied separately.) Such a variable is termed *dense* or *full*. An example might be the PRICE variable, which in every incarnation should have a record for every possible value of the index (item-id). In such a case the characteristic function of such a variable is simply T.

We could use the trivial characteristic function for a computed variable as well, but more (useful) information can be obtained through the application of Theorems 3-6 to the defining HIBOL flow equation. Likewise, we can use Theorems 1 and 2 to obtain useful characteristic functions for critical index sets. Characteristic functions thus obtained are called *one-step characteristic functions*.

It should be easy to see that for any characteristic function if an occurrence of (DEFINED variable) is replaced by the characteristic function for variable, the result will be a logically equivalent characteristic function. This is termed *back-substitution* of characteristic functions. If back-substitution is applied recursively, the result will be a characteristic function containing only

DEFINED's whose arguments are non-computed variables. This is called *total* back-substitution.

Total back-substitution of all characteristic functions has the advantage of making them all into a uniform form, thus facilitating comparison and logical manipulation.

#### **IV.2.3 Example**

Consider the flow equations:

S IS H \* R IF H PRESENT AND R PRESENT

X IS (H - 40) \* R / 2 IF H PRESENT AND R PRESENT AND H > 40

P IS S + X IF S PRESENT AND X PRESENT

ELSE S IF S PRESENT

ELSE X IF X PRESENT

where the flows H and R are system inputs, all flow have the index (key) and all computations are performed daily. The one-step characteristic functions of the necessary input sets are:<sup>28</sup>

$$NIS(S)_{char} = (\text{AND } (\text{DEFINED } (H \text{ DAY key})) \\ (\text{DEFINED } (R \text{ DAY key})))$$

$$NIS(X)_{char} = (\text{AND } (\text{DEFINED } (H \text{ DAY key})) \\ (\text{DEFINED } (R \text{ DAY key})) \\ (\text{GREATERP } (H \text{ DAY key}) 40))$$

$$NIS(P)_{char} = (\text{OR}(\text{DEFINED } (S \text{ DAY key})) \\ (\text{DEFINED } (X \text{ DAY key})))$$

From these we deduce (by Theorem 9) the following results

- I. Computation S can be driven by either H or R, since both

---

<sup>28</sup> We use the outputs as the computation names and drop the level subscript since there is only one level.

$$\text{NIS}(S)_{\text{char}} \rightarrow (\text{DEFINED (H DAY key)}) \quad (1.a)$$

and

$$\text{NIS}(S)_{\text{char}} \rightarrow (\text{DEFINED (R DAY key)}) \quad (1.b)$$

are true

2. Computation X can be driven by either H or R, since both

$$\text{NIS}(X)_{\text{char}} \rightarrow (\text{DEFINED (H DAY key)}) \quad (2.a)$$

and

$$\text{NIS}(X)_{\text{char}} \rightarrow (\text{DEFINED (R DAY key)}) \quad (2.b)$$

are true

3. Computation P must be driven by both S and X, since neither

$$\text{NIS}(P)_{\text{char}} \rightarrow (\text{DEFINED (S DAY key)}) \quad (3.a)$$

nor

$$\text{NIS}(P)_{\text{char}} \rightarrow (\text{DEFINED (X DAY key)}) \quad (3.b)$$

are true, but

$$\text{NIS}(P)_{\text{char}} \rightarrow (\text{OR } (\text{DEFINED (S DAY key)})) \quad (3.c)  
(\text{DEFINED (X DAY key)})$$

is true

However, we know that

$$\text{IS}(S)_{\text{char}} = (\text{AND}(\text{DEFINED (H DAY key)})  
(\text{DEFINED (R DAY key)}))$$

$$\text{IS}(X)_{\text{char}} = (\text{AND}(\text{DEFINED (H DAY key)})  
(\text{DEFINED (R DAY key)}))  
(\text{GREATERP (H DAY key) 40}))$$

so back-substitution of characteristic functions yields

$$\begin{aligned}
 \text{NIS}(P)_{\text{char}} &= (\text{OR } (\text{DEFINED } (\text{S DAY key})) \\
 &\quad (\text{DEFINED } (\text{X DAY key}))) \\
 &= (\text{OR } (\text{AND } (\text{DEFINED } (\text{H DAY key})) \\
 &\quad (\text{DEFINED } (\text{R DAY key}))) \\
 &\quad (\text{AND } (\text{DEFINED } (\text{H DAY key})) \\
 &\quad (\text{DEFINED } (\text{R DAY key})) \\
 &\quad (\text{GREATERP } (\text{H DAY key}) 48))) \\
 &= (\text{AND } (\text{DEFINED } (\text{H DAY key})) \\
 &\quad (\text{DEFINED } (\text{R DAY key})))
 \end{aligned}$$

Thus, formula (3.a)

$$\begin{aligned}
 \text{NIS}(P)_{\text{char}} \rightarrow & (\text{DEFINED } (\text{S DAY key})) \\
 \text{becomes} \\
 & (\text{AND } (\text{DEFINED } (\text{H DAY key})) \text{ } (\text{DEFINED } (\text{R DAY key}))) \\
 \rightarrow \\
 & (\text{AND } (\text{DEFINED } (\text{H DAY key})) \text{ } (\text{DEFINED } (\text{R DAY key})))
 \end{aligned}$$

which is obviously true. Thus, back-substitution has revealed that computation P can be driven by S alone.

## Part V: Loop Implementation (see 02M13201, P11)

(1) (g) (1) YAO XI 03M1301

(1) (g) (1) YAO XI 03M1301 (OMA, P11)

Each (aggregate) computation (job step, program) in the design produced by Protosystem's

(1) (g) (1) YAO XI 03M1301 (OMA, P11)

Optimizing Designer is merely a loop that reads all the records of its input file to generate the

(1) (g) (1) YAO XI 03M1301 (OMA)

records of its output file(s). Implementation of the loop involves writing the appropriate specific

(1) (g) (1) YAO XI 03M1301

control and data structures necessary for this loop and the OMA involved.<sup>29</sup> The main complications

(1) (g) (1) YAO XI 03M1301 (OMA)

that arise in this process stem from the data flows (OMA) of the loops to be implemented and the

hybrid nature of files and loops resulting from aggregation. It is easier to explain <sup>(a)</sup> chapter 2 and T

through a series of examples, beginning with the simplest case, <sup>(b)</sup> the most general case.

(1) (g) (1)  
03M030

In this way the full complexity is visible. Consider first the simple loop <sup>(c)</sup> implementable parts

corresponding to the loops of aggregation, <sup>(d)</sup> chapter 2, <sup>(e)</sup> section 5, <sup>(f)</sup> section 6, <sup>(g)</sup> section 7, <sup>(h)</sup> section 8, <sup>(i)</sup> section 9, <sup>(j)</sup> section 10, <sup>(k)</sup> section 11, <sup>(l)</sup> section 12, <sup>(m)</sup> section 13, <sup>(n)</sup> section 14, <sup>(o)</sup> section 15, <sup>(p)</sup> section 16, <sup>(q)</sup> section 17, <sup>(r)</sup> section 18, <sup>(s)</sup> section 19, <sup>(t)</sup> section 20, <sup>(u)</sup> section 21, <sup>(v)</sup> section 22, <sup>(w)</sup> section 23, <sup>(x)</sup> section 24, <sup>(y)</sup> section 25, <sup>(z)</sup> section 26, <sup>(aa)</sup> section 27, <sup>(bb)</sup> section 28, <sup>(cc)</sup> section 29, <sup>(dd)</sup> section 30, <sup>(ee)</sup> section 31, <sup>(ff)</sup> section 32, <sup>(gg)</sup> section 33, <sup>(hh)</sup> section 34, <sup>(ii)</sup> section 35, <sup>(jj)</sup> section 36, <sup>(kk)</sup> section 37, <sup>(ll)</sup> section 38, <sup>(mm)</sup> section 39, <sup>(nn)</sup> section 40, <sup>(oo)</sup> section 41, <sup>(pp)</sup> section 42, <sup>(qq)</sup> section 43, <sup>(rr)</sup> section 44, <sup>(ss)</sup> section 45, <sup>(tt)</sup> section 46, <sup>(uu)</sup> section 47, <sup>(vv)</sup> section 48, <sup>(ww)</sup> section 49, <sup>(xx)</sup> section 50, <sup>(yy)</sup> section 51, <sup>(zz)</sup> section 52, <sup>(aa)</sup> section 53, <sup>(bb)</sup> section 54, <sup>(cc)</sup> section 55, <sup>(dd)</sup> section 56, <sup>(ee)</sup> section 57, <sup>(ff)</sup> section 58, <sup>(gg)</sup> section 59, <sup>(hh)</sup> section 60, <sup>(ii)</sup> section 61, <sup>(jj)</sup> section 62, <sup>(kk)</sup> section 63, <sup>(ll)</sup> section 64, <sup>(mm)</sup> section 65, <sup>(nn)</sup> section 66, <sup>(oo)</sup> section 67, <sup>(pp)</sup> section 68, <sup>(qq)</sup> section 69, <sup>(rr)</sup> section 70, <sup>(uu)</sup> section 71, <sup>(vv)</sup> section 72, <sup>(ww)</sup> section 73, <sup>(xx)</sup> section 74, <sup>(yy)</sup> section 75, <sup>(zz)</sup> section 76, <sup>(aa)</sup> section 77, <sup>(bb)</sup> section 78, <sup>(cc)</sup> section 79, <sup>(dd)</sup> section 80, <sup>(ee)</sup> section 81, <sup>(ff)</sup> section 82, <sup>(gg)</sup> section 83, <sup>(hh)</sup> section 84, <sup>(ii)</sup> section 85, <sup>(jj)</sup> section 86, <sup>(kk)</sup> section 87, <sup>(ll)</sup> section 88, <sup>(mm)</sup> section 89, <sup>(nn)</sup> section 90, <sup>(oo)</sup> section 91, <sup>(pp)</sup> section 92, <sup>(qq)</sup> section 93, <sup>(rr)</sup> section 94, <sup>(uu)</sup> section 95, <sup>(vv)</sup> section 96, <sup>(ww)</sup> section 97, <sup>(xx)</sup> section 98, <sup>(yy)</sup> section 99, <sup>(zz)</sup> section 100, <sup>(aa)</sup> section 101, <sup>(bb)</sup> section 102, <sup>(cc)</sup> section 103, <sup>(dd)</sup> section 104, <sup>(ee)</sup> section 105, <sup>(ff)</sup> section 106, <sup>(gg)</sup> section 107, <sup>(hh)</sup> section 108, <sup>(ii)</sup> section 109, <sup>(jj)</sup> section 110, <sup>(kk)</sup> section 111, <sup>(ll)</sup> section 112, <sup>(mm)</sup> section 113, <sup>(nn)</sup> section 114, <sup>(oo)</sup> section 115, <sup>(pp)</sup> section 116, <sup>(qq)</sup> section 117, <sup>(rr)</sup> section 118, <sup>(uu)</sup> section 119, <sup>(vv)</sup> section 120, <sup>(ww)</sup> section 121, <sup>(xx)</sup> section 122, <sup>(yy)</sup> section 123, <sup>(zz)</sup> section 124, <sup>(aa)</sup> section 125, <sup>(bb)</sup> section 126, <sup>(cc)</sup> section 127, <sup>(dd)</sup> section 128, <sup>(ee)</sup> section 129, <sup>(ff)</sup> section 130, <sup>(gg)</sup> section 131, <sup>(hh)</sup> section 132, <sup>(ii)</sup> section 133, <sup>(jj)</sup> section 134, <sup>(kk)</sup> section 135, <sup>(ll)</sup> section 136, <sup>(mm)</sup> section 137, <sup>(nn)</sup> section 138, <sup>(oo)</sup> section 139, <sup>(pp)</sup> section 140, <sup>(qq)</sup> section 141, <sup>(rr)</sup> section 142, <sup>(uu)</sup> section 143, <sup>(vv)</sup> section 144, <sup>(ww)</sup> section 145, <sup>(xx)</sup> section 146, <sup>(yy)</sup> section 147, <sup>(zz)</sup> section 148, <sup>(aa)</sup> section 149, <sup>(bb)</sup> section 150, <sup>(cc)</sup> section 151, <sup>(dd)</sup> section 152, <sup>(ee)</sup> section 153, <sup>(ff)</sup> section 154, <sup>(gg)</sup> section 155, <sup>(hh)</sup> section 156, <sup>(ii)</sup> section 157, <sup>(jj)</sup> section 158, <sup>(kk)</sup> section 159, <sup>(ll)</sup> section 160, <sup>(mm)</sup> section 161, <sup>(nn)</sup> section 162, <sup>(oo)</sup> section 163, <sup>(pp)</sup> section 164, <sup>(qq)</sup> section 165, <sup>(rr)</sup> section 166, <sup>(uu)</sup> section 167, <sup>(vv)</sup> section 168, <sup>(ww)</sup> section 169, <sup>(xx)</sup> section 170, <sup>(yy)</sup> section 171, <sup>(zz)</sup> section 172, <sup>(aa)</sup> section 173, <sup>(bb)</sup> section 174, <sup>(cc)</sup> section 175, <sup>(dd)</sup> section 176, <sup>(ee)</sup> section 177, <sup>(ff)</sup> section 178, <sup>(gg)</sup> section 179, <sup>(hh)</sup> section 180, <sup>(ii)</sup> section 181, <sup>(jj)</sup> section 182, <sup>(kk)</sup> section 183, <sup>(ll)</sup> section 184, <sup>(mm)</sup> section 185, <sup>(nn)</sup> section 186, <sup>(oo)</sup> section 187, <sup>(pp)</sup> section 188, <sup>(qq)</sup> section 189, <sup>(rr)</sup> section 190, <sup>(uu)</sup> section 191, <sup>(vv)</sup> section 192, <sup>(ww)</sup> section 193, <sup>(xx)</sup> section 194, <sup>(yy)</sup> section 195, <sup>(zz)</sup> section 196, <sup>(aa)</sup> section 197, <sup>(bb)</sup> section 198, <sup>(cc)</sup> section 199, <sup>(dd)</sup> section 200, <sup>(ee)</sup> section 201, <sup>(ff)</sup> section 202, <sup>(gg)</sup> section 203, <sup>(hh)</sup> section 204, <sup>(ii)</sup> section 205, <sup>(jj)</sup> section 206, <sup>(kk)</sup> section 207, <sup>(ll)</sup> section 208, <sup>(mm)</sup> section 209, <sup>(nn)</sup> section 210, <sup>(oo)</sup> section 211, <sup>(pp)</sup> section 212, <sup>(qq)</sup> section 213, <sup>(rr)</sup> section 214, <sup>(uu)</sup> section 215, <sup>(vv)</sup> section 216, <sup>(ww)</sup> section 217, <sup>(xx)</sup> section 218, <sup>(yy)</sup> section 219, <sup>(zz)</sup> section 220, <sup>(aa)</sup> section 221, <sup>(bb)</sup> section 222, <sup>(cc)</sup> section 223, <sup>(dd)</sup> section 224, <sup>(ee)</sup> section 225, <sup>(ff)</sup> section 226, <sup>(gg)</sup> section 227, <sup>(hh)</sup> section 228, <sup>(ii)</sup> section 229, <sup>(jj)</sup> section 230, <sup>(kk)</sup> section 231, <sup>(ll)</sup> section 232, <sup>(mm)</sup> section 233, <sup>(nn)</sup> section 234, <sup>(oo)</sup> section 235, <sup>(pp)</sup> section 236, <sup>(qq)</sup> section 237, <sup>(rr)</sup> section 238, <sup>(uu)</sup> section 239, <sup>(vv)</sup> section 240, <sup>(ww)</sup> section 241, <sup>(xx)</sup> section 242, <sup>(yy)</sup> section 243, <sup>(zz)</sup> section 244, <sup>(aa)</sup> section 245, <sup>(bb)</sup> section 246, <sup>(cc)</sup> section 247, <sup>(dd)</sup> section 248, <sup>(ee)</sup> section 249, <sup>(ff)</sup> section 250, <sup>(gg)</sup> section 251, <sup>(hh)</sup> section 252, <sup>(ii)</sup> section 253, <sup>(jj)</sup> section 254, <sup>(kk)</sup> section 255, <sup>(ll)</sup> section 256, <sup>(mm)</sup> section 257, <sup>(nn)</sup> section 258, <sup>(oo)</sup> section 259, <sup>(pp)</sup> section 260, <sup>(qq)</sup> section 261, <sup>(rr)</sup> section 262, <sup>(uu)</sup> section 263, <sup>(vv)</sup> section 264, <sup>(ww)</sup> section 265, <sup>(xx)</sup> section 266, <sup>(yy)</sup> section 267, <sup>(zz)</sup> section 268, <sup>(aa)</sup> section 269, <sup>(bb)</sup> section 270, <sup>(cc)</sup> section 271, <sup>(dd)</sup> section 272, <sup>(ee)</sup> section 273, <sup>(ff)</sup> section 274, <sup>(gg)</sup> section 275, <sup>(hh)</sup> section 276, <sup>(ii)</sup> section 277, <sup>(jj)</sup> section 278, <sup>(kk)</sup> section 279, <sup>(ll)</sup> section 280, <sup>(mm)</sup> section 281, <sup>(nn)</sup> section 282, <sup>(oo)</sup> section 283, <sup>(pp)</sup> section 284, <sup>(qq)</sup> section 285, <sup>(rr)</sup> section 286, <sup>(uu)</sup> section 287, <sup>(vv)</sup> section 288, <sup>(ww)</sup> section 289, <sup>(xx)</sup> section 290, <sup>(yy)</sup> section 291, <sup>(zz)</sup> section 292, <sup>(aa)</sup> section 293, <sup>(bb)</sup> section 294, <sup>(cc)</sup> section 295, <sup>(dd)</sup> section 296, <sup>(ee)</sup> section 297, <sup>(ff)</sup> section 298, <sup>(gg)</sup> section 299, <sup>(hh)</sup> section 300, <sup>(ii)</sup> section 301, <sup>(jj)</sup> section 302, <sup>(kk)</sup> section 303, <sup>(ll)</sup> section 304, <sup>(mm)</sup> section 305, <sup>(nn)</sup> section 306, <sup>(oo)</sup> section 307, <sup>(pp)</sup> section 308, <sup>(qq)</sup> section 309, <sup>(rr)</sup> section 310, <sup>(uu)</sup> section 311, <sup>(vv)</sup> section 312, <sup>(ww)</sup> section 313, <sup>(xx)</sup> section 314, <sup>(yy)</sup> section 315, <sup>(zz)</sup> section 316, <sup>(aa)</sup> section 317, <sup>(bb)</sup> section 318, <sup>(cc)</sup> section 319, <sup>(dd)</sup> section 320, <sup>(ee)</sup> section 321, <sup>(ff)</sup> section 322, <sup>(gg)</sup> section 323, <sup>(hh)</sup> section 324, <sup>(ii)</sup> section 325, <sup>(jj)</sup> section 326, <sup>(kk)</sup> section 327, <sup>(ll)</sup> section 328, <sup>(mm)</sup> section 329, <sup>(nn)</sup> section 330, <sup>(oo)</sup> section 331, <sup>(pp)</sup> section 332, <sup>(qq)</sup> section 333, <sup>(rr)</sup> section 334, <sup>(uu)</sup> section 335, <sup>(vv)</sup> section 336, <sup>(ww)</sup> section 337, <sup>(xx)</sup> section 338, <sup>(yy)</sup> section 339, <sup>(zz)</sup> section 340, <sup>(aa)</sup> section 341, <sup>(bb)</sup> section 342, <sup>(cc)</sup> section 343, <sup>(dd)</sup> section 344, <sup>(ee)</sup> section 345, <sup>(ff)</sup> section 346, <sup>(gg)</sup> section 347, <sup>(hh)</sup> section 348, <sup>(ii)</sup> section 349, <sup>(jj)</sup> section 350, <sup>(kk)</sup> section 351, <sup>(ll)</sup> section 352, <sup>(mm)</sup> section 353, <sup>(nn)</sup> section 354, <sup>(oo)</sup> section 355, <sup>(pp)</sup> section 356, <sup>(qq)</sup> section 357, <sup>(rr)</sup> section 358, <sup>(uu)</sup> section 359, <sup>(vv)</sup> section 360, <sup>(ww)</sup> section 361, <sup>(xx)</sup> section 362, <sup>(yy)</sup> section 363, <sup>(zz)</sup> section 364, <sup>(aa)</sup> section 365, <sup>(bb)</sup> section 366, <sup>(cc)</sup> section 367, <sup>(dd)</sup> section 368, <sup>(ee)</sup> section 369, <sup>(ff)</sup> section 370, <sup>(gg)</sup> section 371, <sup>(hh)</sup> section 372, <sup>(ii)</sup> section 373, <sup>(jj)</sup> section 374, <sup>(kk)</sup> section 375, <sup>(ll)</sup> section 376, <sup>(mm)</sup> section 377, <sup>(nn)</sup> section 378, <sup>(oo)</sup> section 379, <sup>(pp)</sup> section 380, <sup>(qq)</sup> section 381, <sup>(rr)</sup> section 382, <sup>(uu)</sup> section 383, <sup>(vv)</sup> section 384, <sup>(ww)</sup> section 385, <sup>(xx)</sup> section 386, <sup>(yy)</sup> section 387, <sup>(zz)</sup> section 388, <sup>(aa)</sup> section 389, <sup>(bb)</sup> section 390, <sup>(cc)</sup> section 391, <sup>(dd)</sup> section 392, <sup>(ee)</sup> section 393, <sup>(ff)</sup> section 394, <sup>(gg)</sup> section 395, <sup>(hh)</sup> section 396, <sup>(ii)</sup> section 397, <sup>(jj)</sup> section 398, <sup>(kk)</sup> section 399, <sup>(ll)</sup> section 400, <sup>(mm)</sup> section 401, <sup>(nn)</sup> section 402, <sup>(oo)</sup> section 403, <sup>(pp)</sup> section 404, <sup>(qq)</sup> section 405, <sup>(rr)</sup> section 406, <sup>(uu)</sup> section 407, <sup>(vv)</sup> section 408, <sup>(ww)</sup> section 409, <sup>(xx)</sup> section 410, <sup>(yy)</sup> section 411, <sup>(zz)</sup> section 412, <sup>(aa)</sup> section 413, <sup>(bb)</sup> section 414, <sup>(cc)</sup> section 415, <sup>(dd)</sup> section 416, <sup>(ee)</sup> section 417, <sup>(ff)</sup> section 418, <sup>(gg)</sup> section 419, <sup>(hh)</sup> section 420, <sup>(ii)</sup> section 421, <sup>(jj)</sup> section 422, <sup>(kk)</sup> section 423, <sup>(ll)</sup> section 424, <sup>(mm)</sup> section 425, <sup>(nn)</sup> section 426, <sup>(oo)</sup> section 427, <sup>(pp)</sup> section 428, <sup>(qq)</sup> section 429, <sup>(rr)</sup> section 430, <sup>(uu)</sup> section 431, <sup>(vv)</sup> section 432, <sup>(ww)</sup> section 433, <sup>(xx)</sup> section 434, <sup>(yy)</sup> section 435, <sup>(zz)</sup> section 436, <sup>(aa)</sup> section 437, <sup>(bb)</sup> section 438, <sup>(cc)</sup> section 439, <sup>(dd)</sup> section 440, <sup>(ee)</sup> section 441, <sup>(ff)</sup> section 442, <sup>(gg)</sup> section 443, <sup>(hh)</sup> section 444, <sup>(ii)</sup> section 445, <sup>(jj)</sup> section 446, <sup>(kk)</sup> section 447, <sup>(ll)</sup> section 448, <sup>(mm)</sup> section 449, <sup>(nn)</sup> section 450, <sup>(oo)</sup> section 451, <sup>(pp)</sup> section 452, <sup>(qq)</sup> section 453, <sup>(rr)</sup> section 454, <sup>(uu)</sup> section 455, <sup>(vv)</sup> section 456, <sup>(ww)</sup> section 457, <sup>(xx)</sup> section 458, <sup>(yy)</sup> section 459, <sup>(zz)</sup> section 460, <sup>(aa)</sup> section 461, <sup>(bb)</sup> section 462, <sup>(cc)</sup> section 463, <sup>(dd)</sup> section 464, <sup>(ee)</sup> section 465, <sup>(ff)</sup> section 466, <sup>(gg)</sup> section 467, <sup>(hh)</sup> section 468, <sup>(ii)</sup> section 469, <sup>(jj)</sup> section 470, <sup>(kk)</sup> section 471, <sup>(ll)</sup> section 472, <sup>(mm)</sup> section 473, <sup>(nn)</sup> section 474, <sup>(oo)</sup> section 475, <sup>(pp)</sup> section 476, <sup>(qq)</sup> section 477, <sup>(rr)</sup> section 478, <sup>(uu)</sup> section 479, <sup>(vv)</sup> section 480, <sup>(ww)</sup> section 481, <sup>(xx)</sup> section 482, <sup>(yy)</sup> section 483, <sup>(zz)</sup> section 484, <sup>(aa)</sup> section 485, <sup>(bb)</sup> section 486, <sup>(cc)</sup> section 487, <sup>(dd)</sup> section 488, <sup>(ee)</sup> section 489, <sup>(ff)</sup> section 490, <sup>(gg)</sup> section 491, <sup>(hh)</sup> section 492, <sup>(ii)</sup> section 493, <sup>(jj)</sup> section 494, <sup>(kk)</sup> section 495, <sup>(ll)</sup> section 496, <sup>(mm)</sup> section 497, <sup>(nn)</sup> section 498, <sup>(oo)</sup> section 499, <sup>(pp)</sup> section 500, <sup>(qq)</sup> section 501, <sup>(rr)</sup> section 502, <sup>(uu)</sup> section 503, <sup>(vv)</sup> section 504, <sup>(ww)</sup> section 505, <sup>(xx)</sup> section 506, <sup>(yy)</sup> section 507, <sup>(zz)</sup> section 508, <sup>(aa)</sup> section 509, <sup>(bb)</sup> section 510, <sup>(cc)</sup> section 511, <sup>(dd)</sup> section 512, <sup>(ee)</sup> section 513, <sup>(ff)</sup> section 514, <sup>(gg)</sup> section 515, <sup>(hh)</sup> section 516, <sup>(ii)</sup> section 517, <sup>(jj)</sup> section 518, <sup>(kk)</sup> section 519, <sup>(ll)</sup> section 520, <sup>(mm)</sup> section 521, <sup>(nn)</sup> section 522, <sup>(oo)</sup> section 523, <sup>(pp)</sup> section 524, <sup>(qq)</sup> section 525, <sup>(rr)</sup> section 526, <sup>(uu)</sup> section 527, <sup>(vv)</sup> section 528, <sup>(ww)</sup> section 529, <sup>(xx)</sup> section 530, <sup>(yy)</sup> section 531, <sup>(zz)</sup> section 532, <sup>(aa)</sup> section 533, <sup>(bb)</sup> section 534, <sup>(cc)</sup> section 535, <sup>(dd)</sup> section 536, <sup>(ee)</sup> section 537, <sup>(ff)</sup> section 538, <sup>(gg)</sup> section 539, <sup>(hh)</sup> section 540, <sup>(ii)</sup> section 541, <sup>(jj)</sup> section 542, <sup>(kk)</sup> section 543, <sup>(ll)</sup> section 544, <sup>(mm)</sup> section 545, <sup>(nn)</sup> section 546, <sup>(oo)</sup> section 547, <sup>(pp)</sup> section 548, <sup>(qq)</sup> section 549, <sup>(rr)</sup> section 550, <sup>(uu)</sup> section 551, <sup>(vv)</sup> section 552, <sup>(ww)</sup> section 553, <sup>(xx)</sup> section 554, <sup>(yy)</sup> section 555, <sup>(zz)</sup> section 556, <sup>(aa)</sup> section 557, <sup>(bb)</sup> section 558, <sup>(cc)</sup> section 559, <sup>(dd)</sup> section 560, <sup>(ee)</sup> section 561, <sup>(ff)</sup> section 562, <sup>(gg)</sup> section 563, <sup>(hh)</sup> section 564, <sup>(ii)</sup> section 565, <sup>(jj)</sup> section 566, <sup>(kk)</sup> section 567, <sup>(ll)</sup> section 568, <sup>(mm)</sup> section 569, <sup>(nn)</sup> section 570, <sup>(oo)</sup> section 571, <sup>(pp)</sup> section 572, <sup>(qq)</sup> section 573, <sup>(rr)</sup> section 574, <sup>(uu)</sup> section 575, <sup>(vv)</sup> section 576, <sup>(ww)</sup> section 577, <sup>(xx)</sup> section 578, <sup>(yy)</sup> section 579, <sup>(zz)</sup> section 580, <sup>(aa)</sup> section 581, <sup>(bb)</sup> section 582, <sup>(cc)</sup> section 583, <sup>(dd)</sup> section 584, <sup>(ee)</sup> section 585, <sup>(ff)</sup> section 586, <sup>(gg)</sup> section 587, <sup>(hh)</sup> section 588, <sup>(ii)</sup> section 589, <sup>(jj)</sup> section 590, <sup>(kk)</sup> section 591, <sup>(ll)</sup> section 592, <sup>(mm)</sup> section 593, <sup>(nn)</sup> section 594, <sup>(oo)</sup> section 595, <sup>(pp)</sup> section 596, <sup>(qq)</sup> section 597, <sup>(rr)</sup> section 598, <sup>(uu)</sup> section 599, <sup>(vv)</sup> section 600, <sup>(ww)</sup> section 601, <sup>(xx)</sup> section 602, <sup>(yy)</sup> section 603, <sup>(zz)</sup> section 604, <sup>(aa)</sup> section 605, <sup>(bb)</sup> section 606, <sup>(cc)</sup> section 607, <sup>(dd)</sup> section 608, <sup>(ee)</sup> section 609, <sup>(ff)</sup> section 610, <sup>(gg)</sup> section 611, <sup>(hh)</sup> section 612, <sup>(ii)</sup> section 613, <sup>(jj)</sup> section 614, <sup>(kk)</sup> section 615, <sup>(ll)</sup> section 616, <sup>(mm)</sup> section 617, <sup>(nn)</sup> section 618, <sup>(oo)</sup> section 619, <sup>(pp)</sup> section 620, <sup>(qq)</sup> section 621, <sup>(rr)</sup> section 622, <sup>(uu)</sup> section 623, <sup>(vv)</sup> section 624, <sup>(ww)</sup> section 625, <sup>(xx)</sup> section 626, <sup>(yy)</sup> section 627, <sup>(zz)</sup> section 628, <sup>(aa)</sup> section 629, <sup>(bb)</sup> section 630, <sup>(cc)</sup> section 631, <sup>(dd)</sup> section 632, <sup>(ee)</sup> section 633, <sup>(ff)</sup> section 634, <sup>(gg)</sup> section 635, <sup>(hh)</sup> section 636, <sup>(ii)</sup> section 637, <sup>(jj)</sup> section 638, <sup>(kk)</sup> section 639, <sup>(ll)</sup> section 640, <sup>(mm)</sup> section 641, <sup>(nn)</sup> section 642, <sup>(oo)</sup> section 643, <sup>(pp)</sup> section 644, <sup>(qq)</sup> section 645, <sup>(rr)</sup> section 646, <sup>(uu)</sup> section 647, <sup>(vv)</sup> section 648, <sup>(ww)</sup> section 649, <sup>(xx)</sup> section 650, <sup>(yy)</sup> section 651, <sup>(zz)</sup> section 652, <sup>(aa)</sup> section 653, <sup>(bb)</sup> section 654, <sup>(cc)</sup> section 655, <sup>(dd)</sup> section 656, <sup>(ee)</sup> section 657, <sup>(ff)</sup> section 658, <sup>(gg)</sup> section 659, <sup>(hh)</sup> section 660, <sup>(ii)</sup> section 661, <sup>(jj)</sup> section 662, <sup>(kk)</sup> section 663, <sup>(ll)</sup> section 664, <sup>(mm)</sup> section 665, <sup>(nn)</sup> section 666, <sup>(oo)</sup> section 667, <sup>(pp)</sup> section 668, <sup>(qq)</sup> section 669, <sup>(rr)</sup> section 670, <sup>(uu)</sup> section 671, <sup>(vv)</sup> section 672, <sup>(ww)</sup> section 673, <sup>(xx)</sup> section 674, <sup>(yy)</sup> section 675, <sup>(zz)</sup> section 676, <sup>(aa)</sup> section 677, <sup>(bb)</sup> section 678, <sup>(cc)</sup> section 679, <sup>(dd)</sup> section 680, <sup>(ee)</sup> section 681, <sup>(ff)</sup> section 682, <sup>(gg)</sup> section 683, <sup>(hh)</sup> section 684, <sup>(ii)</sup> section 685, <sup>(jj)</sup> section 686, <sup>(kk)</sup> section 687, <sup>(ll)</sup> section 688, <sup>(mm)</sup> section 689, <sup>(nn)</sup> section 690, <sup>(oo)</sup> section 691, <sup>(pp)</sup> section 692, <sup>(qq)</sup> section 693, <sup>(rr)</sup> section 694, <sup>(uu)</sup> section 695, <sup>(vv)</sup> section 696, <sup>(ww)</sup> section 697, <sup>(xx)</sup> section 698, <sup>(yy)</sup> section 699, <sup>(zz)</sup> section 700, <sup>(aa)</sup> section 701, <sup>(bb)</sup> section 702, <sup>(cc)</sup> section 703, <sup>(dd)</sup> section 704, <sup>(ee)</sup> section 705, <sup>(ff)</sup> section 706, <sup>(gg)</sup> section 707, <sup>(hh)</sup> section 708, <sup>(ii)</sup> section 709, <sup>(jj)</sup> section 710, <sup>(kk)</sup> section 711, <sup>(ll)</sup> section 712, <sup>(mm)</sup> section 713, <sup>(nn)</sup> section 714, <sup>(oo)</sup> section 715, <sup>(pp)</sup> section 716, <sup>(qq)</sup> section 717, <sup>(rr)</sup> section 718, <sup>(uu)</sup> section 719, <sup>(vv)</sup> section 720, <sup>(ww)</sup> section 721, <sup>(xx)</sup> section 722, <sup>(yy)</sup> section 723, <sup>(zz)</sup> section 724, <sup>(aa)</sup> section 725, <sup>(bb)</sup> section 726, <sup>(cc)</sup> section 727, <sup>(dd)</sup> section 728, <sup>(ee)</sup> section 729, <sup>(ff)</sup> section 730, <sup>(gg)</sup> section 731, <sup>(hh)</sup> section 732, <sup>(ii)</sup> section 733, <sup>(jj)</sup> section 734, <sup>(kk)</sup> section 735, <sup>(ll)</sup> section 736, <sup>(mm)</sup> section 737, <sup>(nn)</sup> section 738, <sup>(oo)</sup> section 739, <sup>(pp)</sup> section 740, <sup>(qq)</sup> section 741, <sup>(rr)</sup> section 742, <sup>(uu)</sup> section 743, <sup>(vv)</sup> section 744, <sup>(ww)</sup> section 745, <sup>(xx)</sup> section 746, <sup>(yy)</sup> section 747, <sup>(zz)</sup> section 748, <sup>(aa)</sup> section 749, <sup>(bb)</sup> section 750, <sup>(cc)</sup> section 751, <sup>(dd)</sup> section 752, <sup>(ee)</sup> section 753, <sup>(ff)</sup> section 754, <sup>(gg)</sup> section 755, <sup>(hh)</sup> section 756, <sup>(ii)</sup> section 757, <sup>(jj)</sup> section 758, <sup>(kk)</sup> section 759, <sup>(ll)</sup> section 760, <sup>(mm)</sup> section 761, <sup>(nn)</sup> section 762, <sup>(oo)</sup> section 763, <sup>(pp)</sup> section 764, <sup>(qq)</sup> section 765, <sup>(rr)</sup> section 766, <sup>(uu)</sup> section 767, <sup>(vv)</sup> section 768, <sup>(ww)</sup> section 769, <sup>(xx)</sup> section 770, <sup>(yy)</sup> section 771, <sup>(zz)</sup> section 772, <sup>(aa)</sup> section 773, <sup>(bb)</sup> section 774, <sup>(cc)</sup> section 775, <sup>(dd)</sup> section 776, <sup>(ee)</sup> section 777, <sup>(ff)</sup> section 778, <sup>(gg)</sup> section 779, <sup>(hh)</sup> section 780, <sup>(ii)</sup> section 781, <sup>(jj)</sup> section 782, <sup>(kk)</sup> section 783, <sup>(ll)</sup> section 784, <sup>(mm)</sup> section 785, <sup>(nn)</sup> section 786, <sup>(oo)</sup> section 787, <sup>(pp)</sup> section 788, <sup>(qq)</sup> section 789, <sup>(rr)</sup> section 790, <sup>(uu)</sup> section 791, <sup>(vv)</sup> section 792, <sup>(ww)</sup> section 793, <sup>(xx)</sup> section 794, <sup>(yy)</sup> section 795, <sup>(zz)</sup> section 796, <sup>(aa)</sup> section 797, <sup>(bb)</sup> section 798, <sup>(cc)</sup> section 799, <sup>(dd)</sup> section 800, <sup>(ee)</sup> section 801, <sup>(ff)</sup> section 802, <sup>(gg)</sup> section 803, <sup>(hh)</sup> section 804, <sup>(ii)</sup> section 805, <sup>(jj)</sup> section 806, <sup

extract the data item (the number of hours worked)

multiply it by 3.00,

assemble the corresponding record of PAY

whose employee-id key is the same as the record read

whose data item's value is the result of multiplying the value of the data item of the record read by 3.00

write the newly created record to the file PAY

To support this iteration, there must be declarations of the data objects to be used

loop initialization

EOF (end-of-file) checking (to terminate the loop)

#### V.I.I.1 Necessary Data Objects and Their Declaration

First there must be declarations for all input and output files. Assume that the files PAY and HOURS are known by these names to the PL/I environment (JCL code can be generated to make this happen). Then the following declarations must appear in the PL/I code:

```
DECLARE HOURS INPUT FILE SEQUENTIAL RECORD,  
          PAY OUTPUT FILE SEQUENTIAL RECORD;
```

There must also be declarations for data structures ancillary to the I/O and control to be performed. In particular, for every input file there must be a record image data structure into which a record of that input can be read. Likewise, for every output file there must be a record image data structure into which a record of that output can be built so that it can be written out. In our simple example, the HOURS and PAY files must have such associated data objects. The PL/I structure can be used for this purpose:

```

DECLARE 1 PAY_RECORD,
        2 EMPLOYEE FIXED DECIMAL (4),
        2 PAY FIXED DECIMAL (4),
1 HOURS_RECORD,
        2 EMPLOYEE FIXED DECIMAL (4),
        2 HOURS FIXED DECIMAL (3);

```

Finally, for each input a flag is needed to indicate the EOF condition for that input. Thus, for the HOURS file we would have the declaration:

```

DECLARE 1 EOF ALIGNED,
        2 HOURS BIT (1) UNALIGNED INITIAL ('0'B);

```

When EOF occurs on the associated file this flag is set to '1'B.

### V.II.2 Loop Initialization

Before iteration all flags must be initialized. This can be done by the use of the INITIAL statement in the declaration (as above for EOF.HOURS). Also all drivers must be read to establish initial values for their indices. In our example, the initialization section would consist of merely:

```
READ FILE (HOURS) INTO (HOURS_RECORD);
```

### V.II.3 EOF Checking and Loop Termination

To detect an EOF condition on a file and set its corresponding flag the PL/I ON construct can be used. For the HOURS file the appropriate code would be:

```
ON ENDFILE (HOURS) EOF.HOURS = '1'B;
```

To enforce iteration termination upon EOF of the driver, the loop is constructed using the form DO WHILE (~ EOF.driver).

#### V.I.1.4 The Loop Itself

Given this supporting structure, the rest of the implementation is easy. The loop itself can be written simply as:

```

DO WHILE (~ EOF.HOURS);
    PAY_RECORD.PAY = HOURS_RECORD.HOURS * 3.0;
    PAY_RECORD.EMPLOYEE = HOURS_RECORD.EMPLOYEE;
    WRITE FILE (PAY) FROM (PAY_RECORD);
    READ FILE (HOURS) INTO (HOURS_RECORD);
END ;

```

When the loop terminates, the job step is ended and the input and output files are automatically closed. The complete PL/I program for the pay calculation computation is given in Fig. 1.

#### V.I.2 Uniform-Index Matching Computations

Let us extend our treatment of single-level loop implementations to those with more than one input. We use as our vehicle the variation of the pay calculation that includes a rate file (indexed by employee-id):

PAY IS RATE \* HOURS IF RATE PRESENT AND HOURS PRESENT

Suppose that the input files RATE and HOURS are to be read sequentially, that their records are sorted by employee-id and that HOURS is used as the loop driver.

Again because the loop is driven by a single input file, it is implemented using the form DO WHILE (~ EOF.driver). However, the computation description dictates that a record of the output file PAY for a given value of the key employee-id is to be produced if and only if there is a record for that employee in HOURS and there is a corresponding record in the RATE file. Therefore, in the body of the loop, before the output record can be calculated, the record (if any) of the non-driving input that matches the current value of the driver's index must be found.

Digitized by Google

### ANSWER

—  
—

卷之三

1000

—  
—

卷之三

卷之三

—  
—

— 1 —

— 1 —

10. The following table shows the number of hours worked by 1000 workers in a certain industry.

10. The following table shows the number of hours worked by 1000 workers in a certain industry.

卷之三

卷之三

卷之三

卷之三

*Continued from back cover*

卷之三

卷之三

卷之三

卷之三

卷之三

卷之三

— 1 —

卷之三

— 1 —

卷之三

卷之三

卷之三

卷之三

Based on many years of research and experience, we have developed tools, policies,

To find the matching record of the non-driving input we read successive records from its file comparing the index value of each record with the current loop index. The general matching algorithm consists of the following loop:

**For each non-driving input:**

1. If FOUND. input is true (indicating that the record currently held in the input's image structure has been used) read the next record of the input.
2. If an EOF condition has occurred on the input, set FOUND. input to false (0) and exit the loop.
3. Otherwise, check the index of the current input record against the index of the current driver record:

If =, set FOUND. input to true and exit.

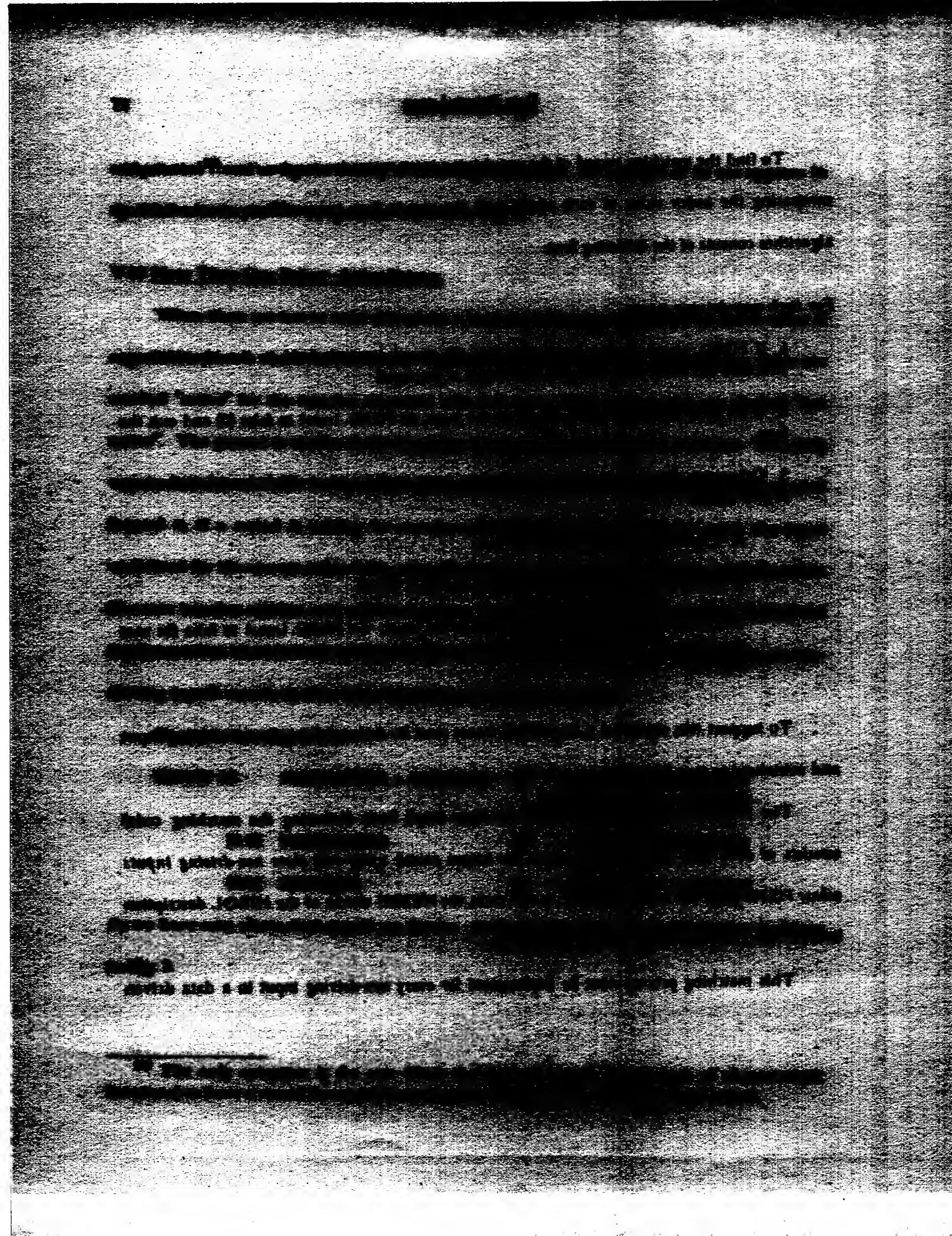
If <, read the next record of the input and go to step 2.

If >, there is no corresponding record in the input. Set FOUND. input to false (in case the index of the record just read may match that of some subsequent driver record) and exit

To support this algorithm a flag FOUND. input must be declared for each non-driving input and initialized to true (1) before the main loop.

The implementation of the rest of the main loop's body (following the matching code) consists of code that attempts to compute the output record using only those non-driving inputs whose FOUND flags are true. Basically, in this code, the PRESENT checks of the HIBOL description become checks on the corresponding FOUND flags.

This matching process must be implemented for every non-driving input in a data driven



## Data Driven Loops

```
PAY_COMP: PROCEDURE;

(declarations)

ON ENDFILE (RATE) EOF.RATE = '1'B;
ON ENDFILE (HOURS) EOF.HOURS = '1'B;

READ FILE (RATE) INTO (RATE_RECORD);
LEVEL_1_MINIMUM.EMPLOYEE = RATE_RECORD.EMPLOYEE;

DO WHILE ( EOF.RATE );
  IF EOF.HOURS
    THEN DO; /* THIS READS ITEMS, SEQUENTIALLY, FROM A FILE UNTIL THE REQUESTED
           RECORD IS FOUND (SET FLAGS TO TRUE) OR PASSED (SET FLAGS TO FALSE). */
    IF FOUND.HOURS_RECORD
      THEN READ FILE (HOURS) INTO (HOURS_RECORD);

    HOURS_RECORD_COMPARE:
    IF EOF.HOURS
      THEN FOUND.HOURS_RECORD = '0'B;
    ELSE IF HOURS_RECORD.EMPLOYEE = LEVEL_1_MINIMUM.EMPLOYEE
      THEN FOUND.HOURS_RECORD = '1'B;
    ELSE IF HOURS_RECORD.EMPLOYEE > LEVEL_1_MINIMUM.EMPLOYEE
      THEN FOUND.HOURS_RECORD = '0'B;
    ELSE DO; READ FILE (HOURS) INTO (HOURS_RECORD);
      GO TO HOURS_RECORD_COMPARE;
    END;
  END;

  IF FOUND.HOURS THEN DO; PAY_RECORD.PAY = RATE_RECORD.RATE * HOURS_RECORD.HOURS;
    PAY_RECORD.EMPLOYEE = LEVEL_1_MINIMUM.EMPLOYEE;
    WRITE FILE (PAY) FROM (PAY_RECORD);
  END;

  READ FILE (RATE) INTO (RATE_RECORD);
  LEVEL_1_MINIMUM.EMPLOYEE = RATE_RECORD.EMPLOYEE;
END;
END PAY_COMP;
```

Figure 2: PL/I code for PAY IS RATE \* HOURS



First, notice that the iteration structure is fundamentally different from that for a single driver loop. The index value determination and EOF checking is now performed at the beginning of the loop body.<sup>31</sup> As always, the iteration is terminated when all drivers are exhausted (when the flag EOF\_SO\_FAR ends up true after all drivers have been read). Thus the loop exit must appear before the output calculations and the form DO WHILE ('1'BY) is used instead of DO WHILE (~EOF, driver) (as in the single driver case). This is just a minor variation on the basic scheme.

What is interesting in the implementation of Fig. 3 is the use of the PL/I ACTIVE structure and the ACTIVE\_DRIVER\_COUNT variable in determining the proper next index value. The idea is to look through the drivers in succession. The first is used to establish a tentative index value for the current iteration. The first driver is also given a number that marks it active (for the time being). If the next driver has the same index value it is given the same number, indicating that it will be active when the first is; if it has a lower index value the loop index is reset and the second driver is assigned a higher number, meaning that it is tentatively active (and, effectively, that the first is inactive). When all drivers have been examined, those sharing the highest ACTIVE number (held in ACTIVE\_DRIVER\_COUNT) are marked defined, and the rest are marked not defined.

## V.2 Multiple-Level Loops

Multiple-level loops introduce the need for maintenance of current index values for each distinct loop level and for control structures to implement loop driving from loops at lower levels.

Multiple-level loops arise from two basic sources: reduction computations and mixed-index matching computations. Let us examine the implementation of each in turn.

---

<sup>31</sup> It could be done at the end of the body if the same code were duplicated as an initialization before the loop were entered. We have refrained from doing this to minimize code.

Digitized by srujanika@gmail.com

1996-1997

Digitized by srujanika@gmail.com

卷之三

10. The following is a list of the names of the members of the family of the author.

—  
—

1996-1997 学年第一学期期中考试

[View Details](#) | [Edit](#) | [Delete](#)

**Figure 1.** A photograph of the experimental setup used to measure the effect of the magnetic field on the rate of diffusion of the tracer in the polymer matrix.

[View Details](#) | [Edit](#) | [Delete](#)

10. The following table summarizes the results of the study. The first column lists the variables, the second column lists the sample size, and the third column lists the estimated effect sizes.

For more information about the study, contact Dr. Michael J. Hwang at (319) 356-4000 or email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

10. The following table shows the number of hours worked by each employee in a company.

10. The following table shows the number of hours worked by each employee in a company.

*Journal of Health Politics, Policy and Law*, Vol. 35, No. 4, December 2010  
DOI 10.1215/03616878-35-4 © 2010 by The University of Chicago

Digitized by srujanika@gmail.com

Figure 1. A photograph of the surface of the ocean showing the effect of the wind on the surface.

10. The following table shows the number of hours worked by 1000 workers in a certain industry.

10. The following table shows the number of hours worked by each employee in a company.

10. The following table shows the number of hours worked by 1000 employees in a company.

For more information about the study, please contact Dr. Michael J. Hwang at (319) 356-4000 or via email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

10. The following table shows the number of hours worked by 1000 employees in a company.

10. The following table shows the number of hours worked by 1000 employees in a company.

For more information about the study, contact Dr. Michael J. Hwang at (319) 356-4000 or email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

## Data Driven Loops

```
ITEMDEMAND_COMP: PROCEDURE;  
  
(declarations)  
  
ON ENDFILE (DEMAND),EOF_DEMAND = '1'B;  
READ FILE (DEMAND) INTO (DEMAND_RECORD);  
IF EOF_DEMAND  
THEN DO; LEVEL_2_MINIMUM.ITEM = DEMAND_RECORD.ITEM;  
      LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;  
      END;  
ELSE LEVEL_1 = '0'B;  
  
DO WHILE (LEVEL_1);  
  IF DEFINED,ITEMDEMAND = '0'B;  
    DO WHILE (LEVEL_2);  
      IF DEFINED,ITEMDEMAND  
      THEN ITEMDEMAND_RECORD.ITEMDEMAND = ITEMDEMAND_RECORD.ITEMDEMAND + DEMAND_RECORD.DEMAND;  
      ELSE DO; ITEMDEMAND_RECORD.ITEMDEMAND = DEMAND_RECORD.DEMAND;  
             DEFINED,ITEMDEMAND = '1'B;  
      END;  
  
      READ FILE (DEMAND) INTO (DEMAND_RECORD);  
      IF EOF_DEMAND  
      THEN DO; LEVEL_2_MINIMUM.ITEM = DEMAND_RECORD.ITEM;  
            IF LEVEL_2_MINIMUM.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM  
            THEN LEVEL_2 = '0'B;  
            END;  
      ELSE DO; LEVEL_2 = '0'B;  
             LEVEL_1 = '0'B;  
      END;  
    END;  
  END;  
  ITEMDEMAND_RECORD.ITEM = LEVEL_1_MINIMUM.ITEM;  
  WRITE FILE (ITEMDEMAND) FROM (ITEMDEMAND_RECORD);  
  
  IF EOF_DEMAND  
  THEN LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;  
  
END;  
END ITEMDEMAND_COMP;
```

Figure 4: PL/I code for ITEMDEMAND IS THE SUM OF DEMAND FOR EACH ITEM-ID

The reader should have little difficulty in understanding this code. Note that the variables LEVEL\_1 and LEVEL\_2 are used as flags to control the iteration of the outer and inner loops, respectively. LEVEL\_1 is essentially equivalent to BTF, but it is more general (e.g., LEVEL\_1 NO  
when the file to be reduced is exhausted). LEVEL\_2 becomes true when the accumulated  
accumulated is exhausted (i.e. when the accumulated value reaches 100% EACH time  
changes value.

The variables LEVEL\_2\_NUMBER, PCH and LEVELS\_1, LEVEL\_2\_NUMBER (PCH) are used to keep track of the current input record's level-1 value and the current record's level-2 value, respectively. A "5" comparison between these two is equivalent to the LEVEL\_2\_NUMBER (PCH) = 5 condition. The variable PCH is used to indicate if the current record is a header record or not. If PCH = 1, then the current record is a header record; otherwise, it is a data record.

**V22 Model 1000 Micro-Computer**

**Comments** [redacted] [redacted] [redacted]

**THE CHURCH IS**

where **EXTENDEDPRICE** and **CURRENTNUMBER** have the index **ITEM-KEY** and **ITEM-KEY** has the index **ITEM-ID**. Suppose that, as above, the **Customer-Number** has qualified that the records of **CURRENTNUMBER** are sorted by the key **ITEM-ID**. As we have shown above, **CURRENTNUMBER** can be used to drive the concatenation.

Because **DIMENSION** is sorted by Item-Id first, the resulting components will be presented as follows:

9. (Initiation) Read a record of the CURRENCYSER file.  
      (1)-NET1 NOAB NOR QMAM30 TO HU2 3BT 21 CHARTERHAT1 net abc> D,0,9,1,8,8,9  
1. Read records from the PRICE file and others.

- a. one is found that has an item-id value matching the driver's item-id value, in which case all EXTENDEDPRICE records for that value can be generated; or
  - b. one is found that has an item-id value greater than the driver's, or the PRICE file is exhausted, in which case there is no matching value and the inner loop can be skipped.
2. (Inner loop) Generate all output records for the given item-id value, reading records from the driver as you go. When a driver record is read that has an item-id value greater than that of the current PRICE record, or the driving file is exhausted, exit.
3. If neither input file is exhausted go to step 1 and repeat; otherwise exit.

In this way each record of the PRICE file is read only once<sup>32</sup>

A PL/I implementation of this algorithm is shown in Fig. 9. The reader will notice that this implementation is unnecessarily inefficient because when a matching PRICE record is not found the inner loop is executed anyway. This is done to illustrate what happens in the general case where there may be calculations in the inner loop that can still be performed without the use of a missing input.

### V.3 Aggregated Computations

The aggregation of two or more computations into one nested loop introduces a consideration not seen before: the synchronization of computations at different loop levels. Consider the two HIBOL computations:

EXTENDEDPRICE IS PRICE \* CURRENTORDER IF      PRICE PRESENT  
AND CURRENTORDER PRESENT

VALUESHIPPED IS PRICE \* ITEMDemand IF      PRICE PRESENT  
AND ITEMDemand PRESENT

---

<sup>32</sup> If CURRENTORDER had been unsorted or sorted differently, records from PRICE would generally be read more than once.



where CURRENTORDER is the same as above (with index (item-id, store-id)) and ITEMDEMAND is a file with index (item-id). As we have seen above, the first computation can be implemented as a two-level nested loop. The second computation iterates over the single key item-id and so has only one level.

When aggregated, the result is a two-level loop:<sup>33</sup>

#### Loop 1 (outer loop)

Level: (item-id)

Inputs: IPRICE, ITEMDEMAND

Prolog: calculate value-shipped

Outputs: empty

Inputs: empty

Epi-log: empty

Outputs: VALUESHIPPED

#### Loop 2 (inner loop)

Level: (item-id, store-id)

Inputs: CURRENTORDER

Prolog: calculate extended-price

Outputs: EXTENDEDPRICE

Inputs: empty

Epi-log: empty

Outputs: empty

What is significant here is that the computations in the aggregate occur in different levels.

Suppose that the PRICE file is guaranteed to have a record for every item-id. Then ITEMDEMAND is the natural choice for a driver for the value-shipped computation because a record of the output will be generated if and only if there is a record in ITEMDEMAND for the same key. As for the extended-price computation, CURRENTORDER is the only possible choice for the driver.

Now the outer loop iterates over item-id values determined by both drivers. Suppose the first record of each driver is read. There are three cases, distinguished by the relative values of the item-id keys in these records:

<sup>33</sup> Notice that in finalized loop description there is no General section.

ANSWER: The answer is 1000. The total number of people in the room is 1000.

Digitized by srujanika@gmail.com

10. The following table shows the number of hours worked by 1000 employees in a company.

10. The following table shows the number of hours worked by 1000 employees.

10. The following table shows the number of hours worked by 1000 employees in a company.

[View Details](#) | [Edit](#) | [Delete](#)

[View Details](#) | [Edit](#) | [Delete](#)

10. The following table shows the number of hours worked by 1000 employees in a company.

For more information about the study, please contact Dr. Michael J. Hwang at (319) 356-4000 or email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

10. The following table shows the number of hours worked by 1000 employees in a company.

[View Details](#) | [Edit](#) | [Delete](#)

10. The following table shows the number of hours worked by 1000 workers in a certain industry.

Digitized by srujanika@gmail.com

For more information about the study, please contact Dr. Michael J. Hwang at (319) 356-4000 or via email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

[View Details](#) | [Edit](#) | [Delete](#)

For more information about the study, please contact Dr. Michael J. Hwang at (310) 794-3000 or via email at [mhwang@ucla.edu](mailto:mhwang@ucla.edu).

10. The following table shows the number of hours worked by 1000 employees in a company.

For more information about the study, please contact Dr. Michael J. Hwang at (310) 794-3000 or via email at [mhwang@ucla.edu](mailto:mhwang@ucla.edu).

For more information about the study, please contact Dr. Michael J. Hwang at (310) 794-3000 or via email at [mhwang@ucla.edu](mailto:mhwang@ucla.edu).

10. The following table shows the number of hours worked by 1000 employees in a company.

10. The following table shows the number of hours worked by 1000 employees in a company.

*Journal of Health Politics, Policy and Law*, Vol. 35, No. 3, June 2010  
DOI 10.1215/03616878-35-3 © 2010 by The University of Chicago

10. The following table summarizes the results of the study. The first column lists the variables, the second column lists the sample size, and the third column lists the estimated effect sizes.

[View Details](#) | [Edit](#) | [Delete](#)

10. The following table shows the number of hours worked by 1000 employees in a company.

## Data Driven Loops

```

(declarations)
(ON conditions)
(read CURRENTORDER and initialize LEVEL_2_MINIMUM.ITEM = CURRENTORDER_RECORD.ITEM)
(read ITEMDEMAND and initialize LEVEL_1_MINIMUM.ITEM = ITEMDEMAND_RECORD.ITEM)
(code to set the synchronization flag for each level to false if its driver had no records)
(comparison of ITEM values to set synchronization flags:
    IF LEVEL_2_MINIMUM.ITEM > LEVEL_1_MINIMUM.ITEM
        THEN DO; DO_LEVEL_2 = '1'B;
        LEVEL_2 = '0'B;
        LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_1_MINIMUM.ITEM;
        END;
    ELSE IF LEVEL_2_MINIMUM.ITEM < LEVEL_1_MINIMUM.ITEM
        THEN DO; DO_LEVEL_1 = '0'B;
        LEVEL_2 = '1'B;
        LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_2_MINIMUM.ITEM;
        END;
    ELSE DO; DO_LEVEL_1 = '1'B;
        LEVEL_2 = '1'B;
        LEVELS_1_THRU_2_MINIMUM.ITEM = LEVEL_1_MINIMUM.ITEM;
        END; )
DO WHILE (LEVEL_1);
    (read PRICE record)
    IF DO_LEVEL_1 THEN (calculate value-shipped) /* Prolog LEVEL_1 */;

    DO WHILE (LEVEL_2);
        IF FOUND.PRICE_RECORD THEN (calculate and write extended-price)
        (read CURRENTORDER and reset LEVEL_2_MINIMUM.ITEM = CURRENTORDER_RECORD.ITEM)
        (check for eof)
        IF LEVEL_2_MINIMUM.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM THEN LEVEL_2 = '0'B;
            ELSE LEVEL_2 = '1'B;
        END /* LEVEL_2 */;

        IF DO_LEVEL_1 THEN DO /* Epilog LEVEL_1 */;
            IF DEFINED:VALUESHIPPED THEN (write value-shipped record)
                (read ITEMDEMAND and reset
                LEVELS_1_MINIMUM.ITEM = ITEMDEMAND_RECORD.ITEM)
            END /* Epilog LEVEL_1 */;

        (synchronization code exactly as above)

    END /* LEVEL_1 */;

```

Figure 6: Illustration of synchronization code for aggregated computations

**enriched oxygen isotopes with respect to standard CDT**

## Data Driven Loops

```
PAY_COMP: PROCEDURE;

DECLARE DSAG1 INPUT FILE SEQUENTIAL RECORD,
      PAY OUTPUT FILE SEQUENTIAL RECORD;
DECLARE 1 PAY_RECORD,
        2 EMPLOYEE FIXED DECIMAL (4),
        2 PAY FIXED DECIMAL (4),
        1 DSAG1_RECORD,
        2 EMPLOYEE FIXED DECIMAL (4),
        2 DEFINED ALIGNED,
          3 HOURS BIT (1),
          3 OVERTIME BIT (1),
        2 HOURS FIXED DECIMAL (3);
        2 OVERTIME FIXED DECIMAL (3);
        2 EMPLOYEE FIXED DECIMAL (4),
        2 HOURS FIXED DECIMAL (3);
DECLARE 1 EOF ALIGNED,
        2 DSAG1 BIT (1) UNALIGNED INITIAL ('0'B);

ON ENDFILE (DSAG1) EOF.DSAG1 = '1'B;

READ FILE (DSAG1) INTO (DSAG1_RECORD);

DO WHILE (~EOF.DSAG1);

  IF DSAG1.DEFINED.HOURS
    THEN DO;

      PAY_RECORD.PAY = DSAG1_RECORD.HOURS * 3.0;

      PAY_RECORD.EMPLOYEE = DSAG1_RECORD.EMPLOYEE;

      WRITE FILE (PAY) FROM (PAY_RECORD);

      READ FILE (DSAG1) INTO (DSAG1_RECORD);

    END;
  ELSE;

    READ FILE (DSAG1) INTO (DSAG1_RECORD);

  END;
END PAY_COMP;
```

Figure 7: PL/I code for PAY IS HOURS \* 3.00 with Aggregated Flow

### V.5.1 Sequential Access

Sequential access of sequentially organized files is the simplest form of organization and is explained. Sequential access of indexed sequentially organized files is also explained. Sequential access of files with regional (C) organization is not possible.

### V.5.2 Core Table Access

When the records of an input file are to be accessed sequentially by a program, code is generated to read them all into core before the processing begins. This generates a PL/I structure that holds not just a single record, but one large enough to hold all the records of sequential in the file. If, for example, the ~~YACI ORDER FILE~~ ~~YACI ORDER FILE~~ above were organized sequentially and were to be accessed by sequential processing, the code would be generated:

```
1 PRICE_RECORD (4000),
2 ITEM_FIXED (40),
2 PRICE_FIXED (40).
```

```
00 PRICE (4000) 00
00 ITEM (40) 00
00 PRICE (40) 00
```

and the code to fill up this table would be:

```
(0.E * 2800H.000000_10A20 - YA9.0D001_YA9)
```

```
00 PRICE_RECORD_INDEX = 1 TO 4000;
READ FILE INPUTFILE INDEX PRICE_RECORD_INDEX INTO (0E00H.000000_10A20) YA9;
IF EOF(PRICE_RECORD_INDEX) THEN EXIT;
DO PRICE_RECORD_INDEX = 1 TO 4000;
    READ FILE INPUTFILE INDEX PRICE_RECORD_INDEX INTO (0E00H.000000_10A20) YA9;
    IF EOF(PRICE_RECORD_INDEX) THEN EXIT;
    DO PRICE_FIXED_INDEX = 1 TO 40;
        READ FILE INPUTFILE INDEX PRICE_FIXED_INDEX INTO (0E00H.000000_10A20) YA9;
    END;
END;
ENDFILE_PRICE: PRICE_RECORD_SIZE = PRICE_RECORD_INDEX - 1;
```

If the input file is sequentially or indexed sequentially organized, the entries in this table are sorted in some order by the record keys. The ~~YACI ORDER FILE~~ ~~YACI ORDER FILE~~ sort order of the input is compatible with the sort order associated with the entries of the sequential ~~YACI ORDER FILE~~ ~~YACI ORDER FILE~~

will be sorted as follows:  $0.E * 2800H.21.YA9 \rightarrow 0E01.YA9$

<sup>25</sup> The only difference is in the JCL definition of the file.

used. If the sort orders are compatible the method of access is completely analogous to sequential access except that "records" are "read" from the table instead of secondary storage (see Fig. 8).

If the input file is "randomly" organized (regional (2)) the access code generates a hash index and then mimics the PL/I access procedure: compare the key values of the indicated table entry with the desired ones; if identical stop; otherwise examine successive entries in wrap-around fashion until an empty slot is found (end of the bucket) or a complete cycle has been made. If the sort orders are not compatible a more complicated binary search is implemented.

#### V.5.3 Random Access

When the records of an input are directly (regional (2)) organized the file is randomly accessed. Instead of using a loop, as with sequential access, a single read, using a calculated key is executed. For example, if the PRICE file in the EXTENDEDPRICE computation (above) were randomly accessed, the accessing part of the code would be:

```
PRICE_RECORD_HASH_VALUE = MOD (5 * (MOD (LEVEL_2_MINIMUM.ITEM,)),);  
PRICE_RECORD_HASH_VALUE_STRING = PRICE_RECORD_HASH_VALUE;  
PRICE_RECORD_HASH_KEY =  
    LEVEL_2_MINIMUM.ITEM || PRICE_RECORD_HASH_VALUE_STRING;  
FOUND.PRICE_RECORD = '1'B;  
READ FILE (PRICE) INTO (PRICE_RECORD) KEY (PRICE_RECORD_HASH_KEY);
```

The first three statements calculate the source key string which has two parts: the region number (rightmost 8 characters) and the comparison key (the remaining characters). The case where the record is not present is handled by the statement:

```
ON KEY (PRICE) IF ONCODE = 51 THEN FOUND.PRICE_RECORD = '0'B;
```

which resets the FOUND flag if a "keyed record not found" error occurs.

## Data Driven Loops

```
IF EOF.PRICE  
THEN DO; IF FOUND.PRICE_RECORD  
    THEN IF PRICE_RECORD_INDEX <= PRICE_RECORD_SIZE  
        THEN PRICE_RECORD_INDEX = PRICE_RECORD_INDEX + 1;  
    ELSE EOF.PRICE = '1'8;  
  
PRICE_RECORD_COMPARE:  
    IF EOF.PRICE  
        THEN FOUND.PRICE_RECORD = '0'8;  
    ELSE IF PRICE_RECORD.ITEM = LEVELS_1_THRU_2_MINIMUM.ITEM  
        THEN FOUND.PRICE_RECORD = '1';  
    ELSE IF PRICE_RECORD.ITEM > LEVELS_1_THRU_2_MINIMUM.ITEM  
        THEN FOUND.PRICE_RECORD = '0'8;  
    ELSE DO; IF FOUND.PRICE_RECORD  
        THEN IF PRICE_RECORD_INDEX <= PRICE_RECORD_SIZE  
            THEN PRICE_RECORD_INDEX = PRICE_RECORD_INDEX + 1;  
        ELSE EOF.PRICE = '1'8;  
  
    GO TO PRICE_RECORD_COMPARE;  
END;  
END;
```

Figure 8: PL/I Code for Reading PRICE by Core Table in the Extended Price Computation

### V.6 The General Case--A Summary

We have seen that the basic code structure for a computation consists of the following four parts:<sup>35</sup>

declarations

on-conditions

loop initialization

the nested loop<sup>36</sup>

The basic structure of the body of each loop in the nested loop is as follows:

read & match non-driving inputs

Prolog calculations

inner loop (if any)

Epilog calculations

write outputs

read active drivers

determine new active drivers

and index values for the next iteration

loop synchronization code

exit on EOF or (for inner loop) sub-index change

---

<sup>35</sup> It may be interesting to note that ProtoSystem I's code generator generates these sections simultaneously as four separate output streams (rather than sequentially) that are concatenated together when they are all finished.

<sup>36</sup> There is no clean-up code following the loop because the end of the job step which is the computation does everything necessary, including the closing of files.

### Appendix I: The Simple Expositional Artificial Language (SEAL)

As an aid to discussing loops we invent an artificial language similar in form to traditional high-level languages such as ALCOL, PL/I and FORTRAN. The basic constructs of this language are:

Iteration: expressed by the construct:

```
for each <loop-index> from <driving-flow-set>
    <body>
end
```

which has the meaning: perform the actions contained in the <body> for each value of the <loop-index> obtained from the flows in the <driving-flow-set>. <loop-index> is either the name of the index associated with the flows in the <driving-flow-set> or (for reasons that become evident in this paper) a sub-index of corresponding sub-flows. The set of values that the <loop-index> takes on is the union of the index sets of the drivers. This set is enumerated at execution time by reading successive records of the drivers.

I/O and defined: input (record fetching) is expressed by the get operator, thus:

```
get <variable-instance>
```

where <variable-instance> specifies a flow and a particular value for its index, represented as a variable (see below). A statement like this means: fetch the indicated record if it exists.

Output is expressed by the write operator, similarly:

```
write <variable-instance>
```

The defined operator is a logical operator for use in conditional expressions. It is applicable only to flow variable instances. The form

```
defined [<variable-instance>]
```

evaluates to "true" if the specified record or the indicated flow exists. In particular, if the record is an input (obtained through a get) it is "defined" if and only if the get succeeded; if the record is an output it is "defined" if and only if the generating code produced a datum for the record.

Conditional Execution: expressed by the familiar if-then-else construct:

```
if <condition> then <statement-list>1
    else <statement-list>2
```

which means that if the logical expression <condition> evaluates to "true" perform the statements in <statement-list><sub>1</sub>; otherwise, perform the statements in <statement-list><sub>2</sub>.

Logical expressions can be formed using the arithmetic comparison operators, the defined operator, and the logical connectives and, or and not.

Conditional Expressions: expressed by the construct:

```
if <condition> then <expression>1
    else <expression>2
```

which evaluates to the value of <expression><sub>1</sub> if the logical expression <condition> evaluates to "true" and to the value of <expression><sub>2</sub> otherwise.

Variables and Assignment: expressed by the construct:

```
<variable> = <expression>
```

where = is the assignment operator.

A variable can be either a scalar or an indexed variable. Flows are represented as indexed variables with an index identical to the flow's index. Thus, DEMAND(item-id, store-id) is the variable corresponding to the DEMAND flow and an instance of its index selects the datum of the corresponding flow record. That is, for example, the statement:

```
DEMAND(1234, 5678) = CURRENTORDER(1234, 5678) +
    BACKORDER(1234, 5678)
```

means that the datum of the record of DEMAND for item #1234 ordered by store #5678 is to get the value obtained by adding the data of the corresponding records from CURRENTORDER and BACKORDER.

Typically, the record-by-record computation implied by a HIBOL flow equation would look like that equation translated into our artificial language (with a generalized index), such as

```

DEMAND(item-id, store-id) =
  if defined(CURRENTORDER(item-id, store-id))
    and defined(BACKORDER(item-id, store-id))
      then CURRENTORDER(item-id, store-id) +
           BACKORDER(item-id, store-id)
    else if defined(CURRENTORDER(item-id, store-id))
      then CURRENTORDER(item-id, store-id)
    else if defined(BACKORDER(item-id, store-id))
      then BACKORDER(item-id, store-id)
    else undefined
  
```

and would appear somewhere in the body of loop.

Sub-flows: A sub-flow (for use in the for each construct) is expressed by:

<flow-variable>(<sub-index>)

For example,

CURRENTORDER(item-id)

denotes the sub-flow of CURRENTORDER consisting of just those records whose indices correspond to the value of the sub-index (item-id). Generally, the value of the indicated sub-index is fixed by an enclosing loop.

References

1. Baron, Robert V., "Structural Analysis in a Very High Level Language", Master's thesis, MIT, 1977.
2. Fleischer, Richard C., "Loop Merger in ProtoSystem I", Bachelor's thesis, MIT, 1978.
3. Ruth, Gregory R., "ProtoSystem I--An Automatic Programming System Prototype", Proceedings of the National Computer Conference, 1978.

Index

- access methods, 16, 86
- active driver**, 74
- aggregated computations**, II, 81
- aggregated flows, 84
- back-substitution**, 64
- characteristic function**, 58, 59
- code generation, 68
- computation, 7
- computation aggregation, 28, 81
- core table access, 88
- correspondence, 2, 14, 19, 71
- critical index set, 9, 47
- datum**, 1
- DEFINED**, 60
- defined, 92
- dense, 64
- driving flow, 10
- driving flow set, 10, 46
- end-of-file, 21, 22, 26, 69
- EOF, 69
- epilog**, 29
- epilog section, 29
- FE-HIBOL**, 6
- file, I, 7, II, 16, 44, 68
- flow**, 1
- flow equation, 3
- flow expression, 2
- for each, 92
- FOR-SOME**, 60
- fundamental driving constraint, 46
- general section, 29
- get, 92
- HIBOL**, I
- index, 1
- index set, 9
- index set of a flow, 9, 47
- injection, 47
- input flow, 8
- iteration set, 4
- key**, 1
- key-tuple, 1
- level compatibility, 36
- loop, II
- loop aggregability, 36
- loop body, II
- loop implementation, 68
- loop level, II, 29
- loop merging, 41
- loop synchronization, 81
- loop-index, II
- matching algorithm, 73
- matching computation, 14, 71
- minimal driving flow set, 58
- mixed indices, 17
- necessary index set, 55
- nested loop structure, II
- non-totally-nested loops, 42
- one-step characteristic function, 64
- Optimizing Designer, 7
- ordering constraints, 36, 40
- output flow, 8
- period, 61
- predicate, 58
- PRESENT**, 2
- projection, 47
- prolog, 29
- prolog section, 29
- random access, 16, 89
- record, 1
- reduction computation, 24, 78
- reduction operators, 2
- restriction, 47
- safe, 50
- SEAL**, 92

**sequential access**, 16, 88  
**simple arithmetic flow expression**, 50  
**simple computation**, 12, 68, 68  
**single-level loop**, 11, 68  
**sub-flow**, 22, 80, 94  
**system input**, 64

**total back-substitution**, 65  
**totally nested loop**, 11

**variable**, 61  
**variable reference**, 61  
**variable specification**, 61

**write**, 92